Teresa M. Corbin (SBN 132360)
Thomas Mavrakakis (SBN 177927)
HOWREY SIMON ARNOLD & WHITE, LLP
301 Ravenswood Avenue
Menlo Park, California 94025
Telephone: (650) 463-8100
Facsimile: (650) 463-8400

Attorneys for Plaintiff SYNOPSYS, INC.
and for Defendants AEROFLEX INCORPORATED,
AMI SEMICONDUCTOR, INC., MATROX
ELECTRONIC SYSTEMS, LTD., MATROX
GRAPHICS, INC., MATROX INTERNATIONAL
CORP. and MATROX TECH, INC.

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

SAN FRANCISCO DIVISION

RICOH COMPANY, LTD., Plaintiff, vs. AEROFLEX INCORPORATED, et al., Defendants.	Case No. C03-04669 MJJ (EMC) Case No. C03-2289 MJJ (EMC) **DECLARATION OF THOMAS C. MAVRAKAKIS IN SUPPORT OF RESPONSIVE CLAIM CONSTRUCTION BRIEF FOR U. S. PATENT NO. 4,922,432**
SYNOPSYS, INC., Plaintiff, vs. RICOH COMPANY, LTD., a Japanese corporation Defendant.	Date: October 29, 2004 Time: 9:30 AM Courtroom: 11 Judge: Martin J. Jenkins

I, Thomas C. Mavrakakis, declare under penalty of perjury as follows:

1. I am an attorney with the law firm of Howrey Simon Arnold & White, LLP, counsel for plaintiff Synopsys, Inc. and Defendants Aeroflex, Incorporated, AMI Semiconductor, Inc., Matrox Electronic Systems, Ltd., Matrox Grapics, Inc., Matrox International Corp. and Matrox Tech, Inc. (collectively, "Defendants"). The following facts are within my personal knowledge and, if called and sworn as a witness, I could and would testify competently thereto.

2. Attached hereto as Exhibit 1 is a true and correct copy of U.S. Patent No. 4,922,432 entitled "Knowledge Based Method And Apparatus For Designing Integrated Circuits Using Functional Specifications", bearing identification numbers DEF012564 – DEF012585.

3. Attached hereto as Exhibit 2 is a true and correct copy of *International Journal of Computer Aided VLSI Design*, edited by George W. Zobrist, bearing identification numbers KBSC001109 – KBSC001131.

4. Attached hereto as Exhibit 3 is a true and correct copy of U.S. Patent No. 5,197,016 entitled "Integrated Silicon-Software Compiler", bearing identification numbers DEF017265 – DEF017284.

5. Attached hereto as Exhibit 4 is a true and correct copy of the File History for U.S. Patent No. 4,922,432, bearing identification numbers DEF011820 – DEF012110.

6. Attached hereto as Exhibit 5 is a true and correct copy of U.S. Patent No. 4,703,435 entitled "Logic Synthesizer", bearing identification numbers DEF012503 – DEF012518.

7. Attached hereto as Exhibit 6 is a true and correct copy of U.S. Patent No. 4,656,603 entitled "Schematic Diagram Generating System Using Library of General Purpose Interactively Selectable Graphic Primitives To Create Special Applications Icons", bearing identification numbers DEF017456 – DEF017482.

8. Attached hereto as Exhibit 7 is a true and correct copy of *IC Mask Design, Essential Layout Techniques*, authored by Christopher Saint, et al., bearing identification numbers DEF085303 – DEF085329.

9. Attached hereto as Exhibit 8 is a true and correct copy of *Microchip Fabrication, A Practical Guide to Semiconductor Processing*, authored by Peter Van Zant, bearing identification numbers DEF085330 – DEF085343.

10. Attached hereto as Exhibit 9 is a true and correct copy of *CAD Tool Integration For ASIC Design: An End-Users Perspective*, authored by Pankaj Kutkal, et al., bearing identification numbers KBSC000031 – KBSC000036.

11. Attached hereto as Exhibit 10 is a true and correct copy of excerpts from *IBM Dictionary of Computing* (1994), bearing identification numbers DEF083932, DEF083936, DEF084073, DEF084074, DEF084423, DEF084535.

12. Attached hereto as Exhibit 11 is a true and correct copy of excerpts from *Webster's Ninth New Collegiate Dictionary* (1987), bearing identification numbers DEF085290 – DEF085298.

13. Attached hereto as Exhibit 12 is a true and correct copy of *CAD Systems for IC Design*, authored by Marvin E. Daniel, et al., bearing identification numbers DEF011951 – DEF011961.

14. Attached hereto as Exhibit 13 is a true and correct copy of *The CMU Design Automation System*, authored by A. Parker, et al., bearing identification numbers DEF012045 – DEF012052.

15. Attached hereto as Exhibit 14 is a true and correct copy of excerpts from *Artificial Intelligence Terminology* (1989), bearing identification numbers DEF079212, DEF079218, DEF079232, DEF079236, DEF079279, DEF079312, DEF079313, DEF079353, DEF079366, DEF079430, DEF079449, DEF079507.

16. Attached hereto as Exhibit 15 is a true and correct copy of excerpts from *Understanding Expert Systems*, authored by Louis E. Frenzel, Jr., bearing identification numbers DEF079512, DEF079516, DEF079525 – DEF079528, DEF079547, DEF079548, DEF079558, DEF079560, DEF079592 – DEF079596, DEF079616 – DEF079627.

17. Attached hereto as Exhibit 16 is a true and correct copy of excerpts from *Microsoft Press Computer Dictionary* (1991), bearing identification numbers DEF083323 – DEF083325.

18. Attached hereto as Exhibit 17 is a true and correct copy of excerpts from *An Artificial Intelligence Approach to VLSI Design*, authored by Thaddeus J. Kowalski, bearing identification numbers DEF078425, DEF078432, DEF078449 – DEF078455.

19. Attached hereto as Exhibit 18 is a true and correct copy of *Expert Systems A Non-Programmer’s Guide to Development and Applications*, authored by Paul Siegel, bearing identification numbers DEF082264, DEF082270, DEF082283 – DEF082285, DEF082291.

20. Attached hereto as Exhibit 19 is a true and correct copy of excerpts from *Expert Systems Principles and Case Studies*, edited by Richard Forsyth, bearing identification numbers DEF079753, DEF079756, DEF079757, DEF079775, DEF079776.

21. Attached hereto as Exhibit 20 is a true and correct copy of *AI in the 1980s and Beyond*, edited by W. Eric L. Grimson, et al, bearing identification numbers DEF083251 – DEF083284.

22. Attached hereto as Exhibit 21 is a true and correct copy of excerpts from *Expert Systems Tools & Applications*, authored by Harmon, et al., bearing identification numbers DEF079985, DEF079992, DEF080261.

I declare under penalty of perjury under the laws of the United States that the foregoing is true and correct and that this declaration was executed at Menlo Park, California, on September 10, 2004.

____
Thomas C. Mavrakakis

United States Patent [19]
Kobayashi et al.

[11] **Patent Number:** **4,922,432**
[45] **Date of Patent:** **May 1, 1990**

[54] KNOWLEDGE BASED METHOD AND APPARATUS FOR DESIGNING INTEGRATED CIRCUITS USING FUNCTIONAL SPECIFICATIONS

[75] Inventors: **Hideaki Kobayashi**, Columbia, S.C.; **Masahiro Shindo**, Osaka, Japan

[73] Assignees: **International Chip Corporation**, Columbia, S.C.; **Ricoh Company, Ltd.**, Tokyo, Japan

[21] Appl. No.: **143,821**

[22] Filed: **Jan. 13, 1988**

[51] **Int. Cl.**⁵ **G06F 15/60**
[52] **U.S. Cl.** **364/490; 364/489; 364/488; 364/521**
[58] **Field of Search** **364/488–491, 364/521, 300, 513**

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,635,208	1/1987	Coleby et al.	364/491
4,638,442	1/1987	Bryant et al.	364/489
4,648,044	3/1987	Hardy et al.	364/513
4,651,284	3/1987	Watanabe et al.	364/491
4,656,603	4/1987	Dunn	364/488
4,658,370	4/1987	Erman et al.	364/513
4,675,829	6/1987	Clemenson	364/513
4,700,317	10/1987	Watanabe et al.	364/521
4,703,435	10/1987	Darringer et al.	364/488
4,803,636	2/1989	Nishiyama et al.	364/491

FOREIGN PATENT DOCUMENTS

1445914	8/1976	United Kingdom	364/490

OTHER PUBLICATIONS

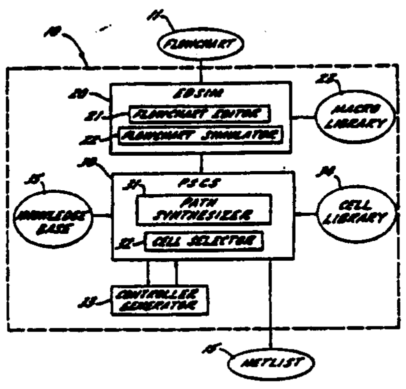
"Verifying Compiled Silicon", by E. K. cheng, VLSI Design, Oct. 1984, pp. 1–4.
"CAD System for IC Design", by M. E. Daniel et al., IEEE Trans. on Computer-Aided Design of Integrated Circuits & Systems, vol. CAD-1, No. 1, Jan. 1982, pp. 2–12.
"An Overview of Logic Synthesis System", by L. Trevillyan, 24th ACM/IEEE Design Automation Conference, 1978, pp. 166–172.
"Methods Used in an Automatic Logic Design Generator", by T. D. Friedman et al., IEEE Trans. on Computers, vol. C-18, No. 7, Jul. 1969, pp. 593–613.
"Experiments in Logic Synthesis", by J. A. Darringer, IEEE ICCC, 1980.
"A Front End Graphic Interface to First Silicon Compiler", by J. H. Nash, EDA 84, Mar. 1984.
"quality of Designs from An Automatic Logic Generator", by T. D. Friedman et al., IEEE 7th DA Conference, 1970, pp. 71–89.
"A New Look at Logic Synthesis", by J. A. Darringer et al., IEEE 17th D. A. Conference 1980, pp. 543–548.
Trevillyan—Trickey, H., Flamel: *A High Level Hardward Compiler*, IEEE Transactions On Computer Aided Design, Mar. 1987, pp. 259–269.
Parker et al., The *CMU Design Automation System—An Example of Automated Data Path Design*, Proceedings Of The 16th Design Automation Conference, Las Vegas, Nev., 1979, pp. 73–80.
An Engineering Approach to Digital Design, William I. Fletcher, Prentice-Hall, Inc., pp. 491–505.

Primary Examiner—Felix D. Gruber
Assistant Examiner—V. N. Trans
Attorney, Agent, or Firm—Bell, Seltzer, Park & Gibson

[57] **ABSTRACT**

The present invention provides a computer-aided design system and method for designing an application specific integrated circuit which enables a user to define functional architecture independent specifications for the integrated circuit and which translates the functional architecture independent specifications into the detailed information needed for directly producing the integrated circuit. The functional architecture independent specifications of the desired integrated circuit can be defined at the functional architecture independent level in a flowchart format. From the flowchart, the system and method uses artificial intelligence and expert systems technology to generate a system controller, to select the necessary integrated circuit hardware cells needed to achieve the functional specifications, and to generate data and control paths for operation of the integrated circuit. This list of hardware cells and their interconnection requirements is set forth in a netlist. From the netlist it is possible using known manual techniques or existing VLSI CAD layout systems to generate the detailed chip level topological information (mask data) required to produce the particular application specific integrated circuit.

20 Claims, 12 Drawing Sheets



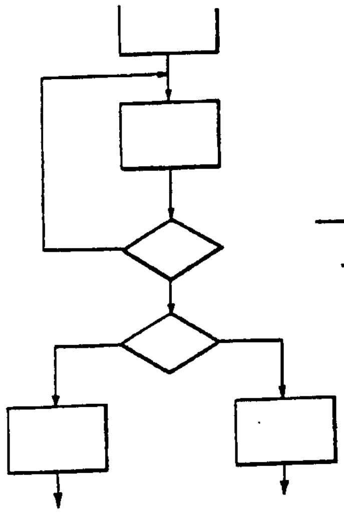


Fig. 1a.
FUNCTIONAL LEVEL

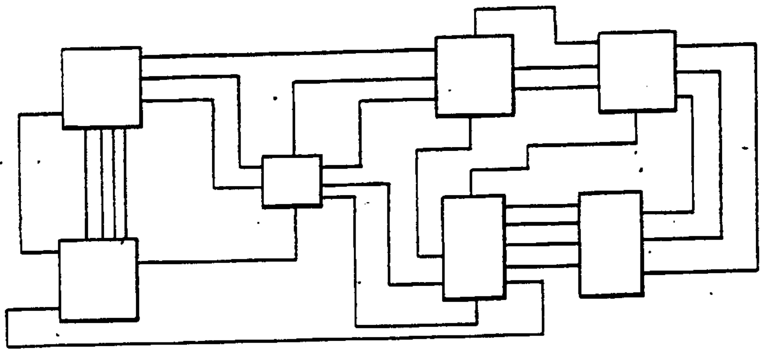


Fig. 1b.
STRUCTURAL LEVEL

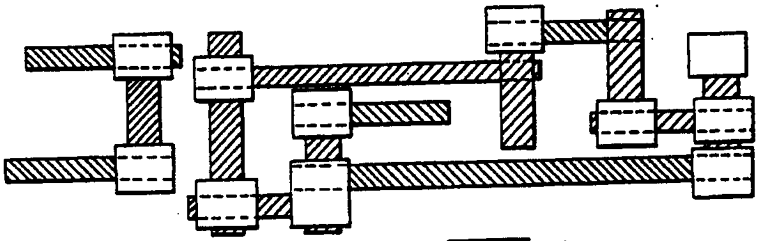


Fig. 1c.
PHYSICAL LAYOUT LEVEL

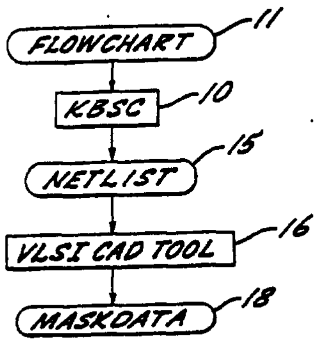

FLOWCHART
11
KBSC
10
NETLIST
15
VLSI CAD TOOL
16
MASKDATA
18

FIG. 2.

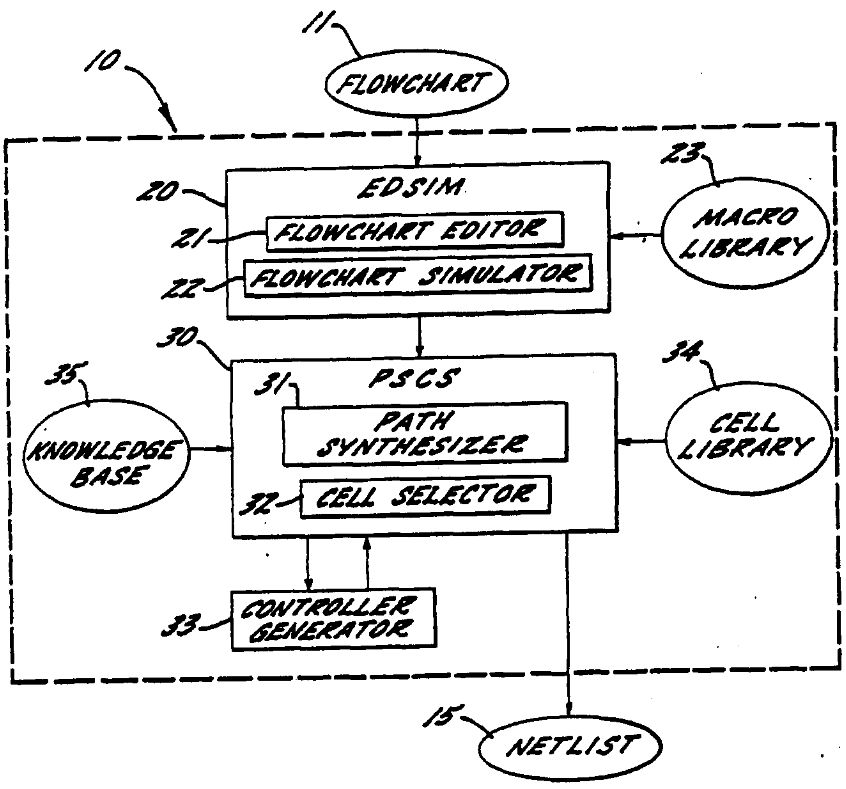

11
10
FLOWCHART
EDSIM
20
21
FLOWCHART EDITOR
22
FLOWCHART SIMULATOR
23
MACRO LIBRARY
30
PSCS
31
PATH SYNTHESIZER
32
CELL SELECTOR
35
KNOWLEDGE BASE
34
CELL LIBRARY
33
CONTROLLER GENERATOR
15
NETLIST

FIG. 3.

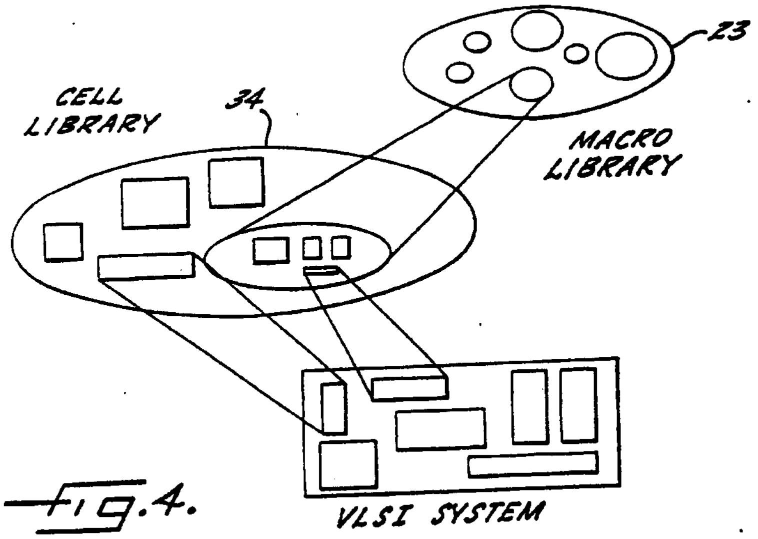

23
CELL LIBRARY
34
MACRO LIBRARY
VLSI SYSTEM

FIG. 4.

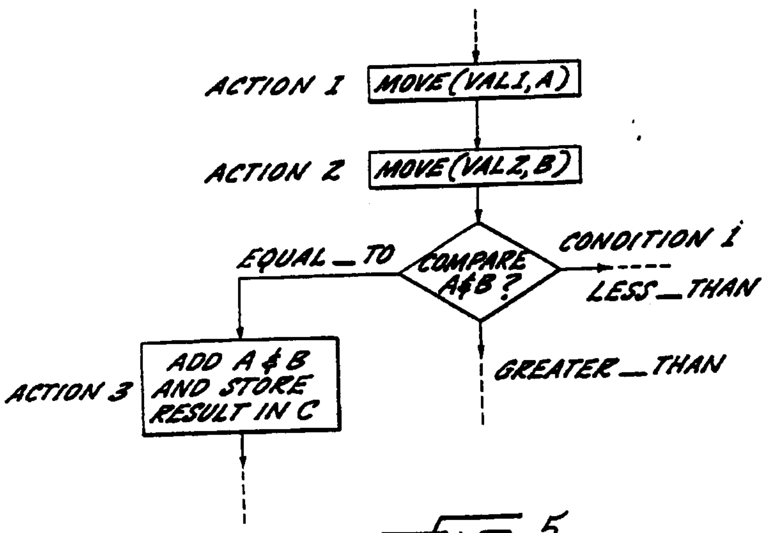

ACTION 1
MOVE(VAL1,A)
ACTION 2
MOVE(VAL2,B)
EQUAL_TO
COMPARE A&B?
CONDITION 1
LESS_THAN
GREATER_THAN
ACTION 3
ADD A & B AND STORE RESULT IN C

FIG. 5.

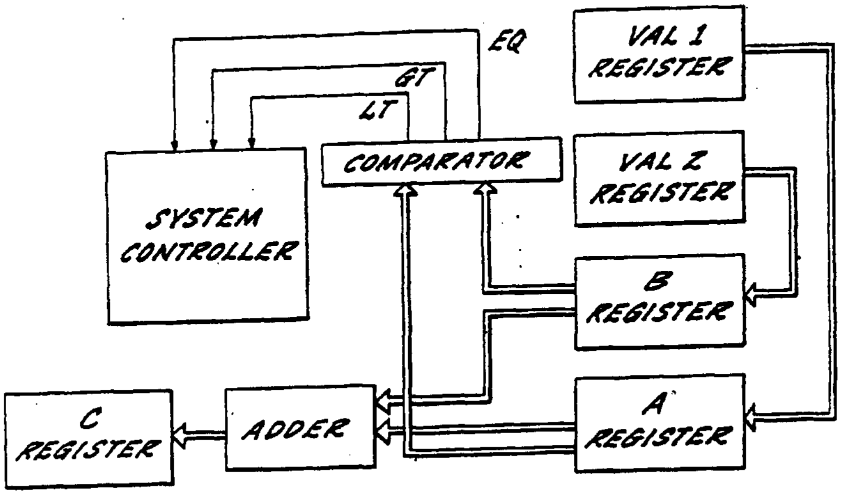

EQ
GT
LT
VAL 1 REGISTER
VAL 2 REGISTER
COMPARATOR
SYSTEM CONTROLLER
B REGISTER
A REGISTER
C REGISTER
ADDER

FIG. 6.

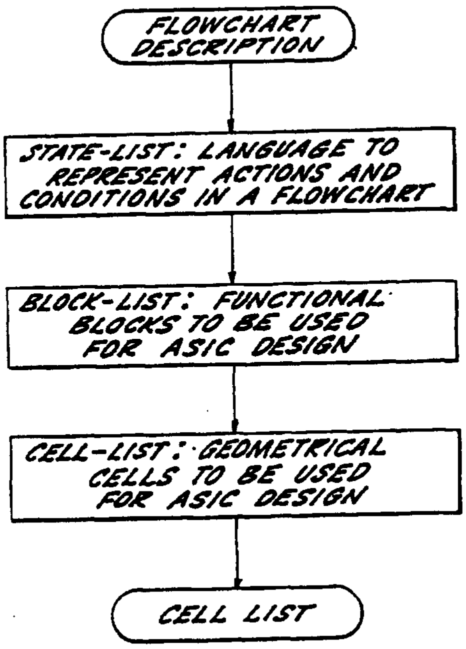

FLOWCHART DESCRIPTION
STATE-LIST: LANGUAGE TO REPRESENT ACTIONS AND CONDITIONS IN A FLOWCHART
BLOCK-LIST: FUNCTIONAL BLOCKS TO BE USED FOR ASIC DESIGN
CELL-LIST: GEOMETRICAL CELLS TO BE USED FOR ASIC DESIGN
CELL LIST

FIG. 9.

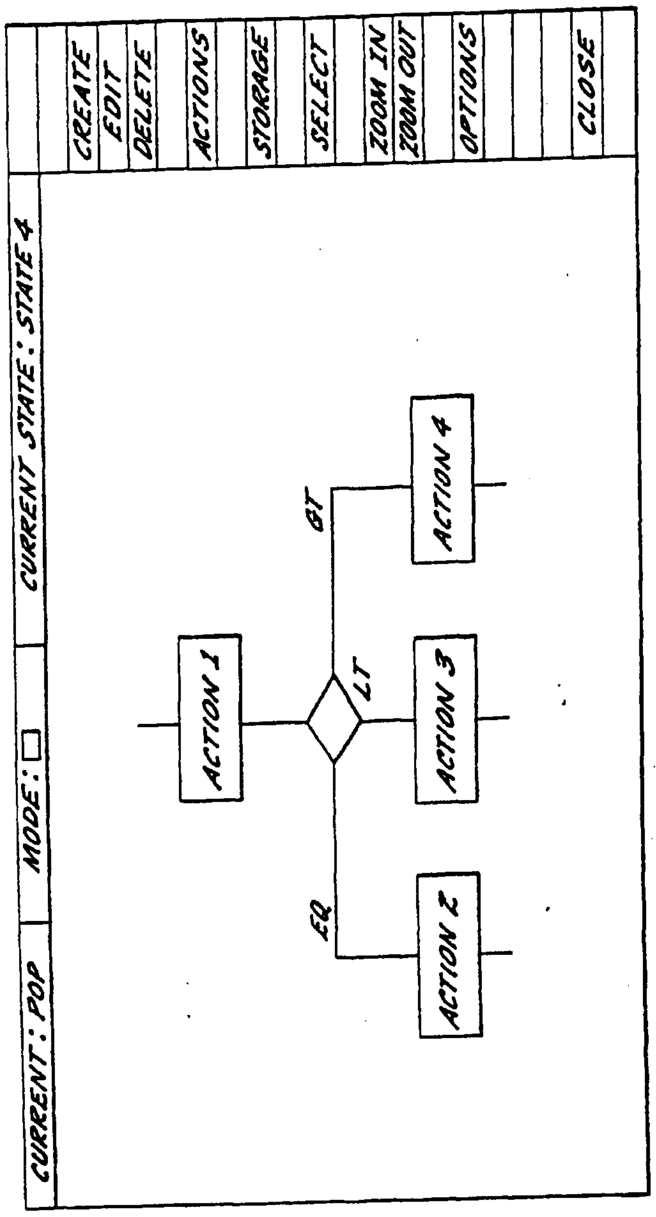

CREATE
EDIT
DELETE
ACTIONS
STORAGE
SELECT
ZOOM IN
ZOOM OUT
OPTIONS
CLOSE
CURRENT STATE: STATE 4
MODE: □
CURRENT: POP
ACTION 1
ACTION 2
ACTION 3
ACTION 4
GT
LT
EQ

FIG. 7.

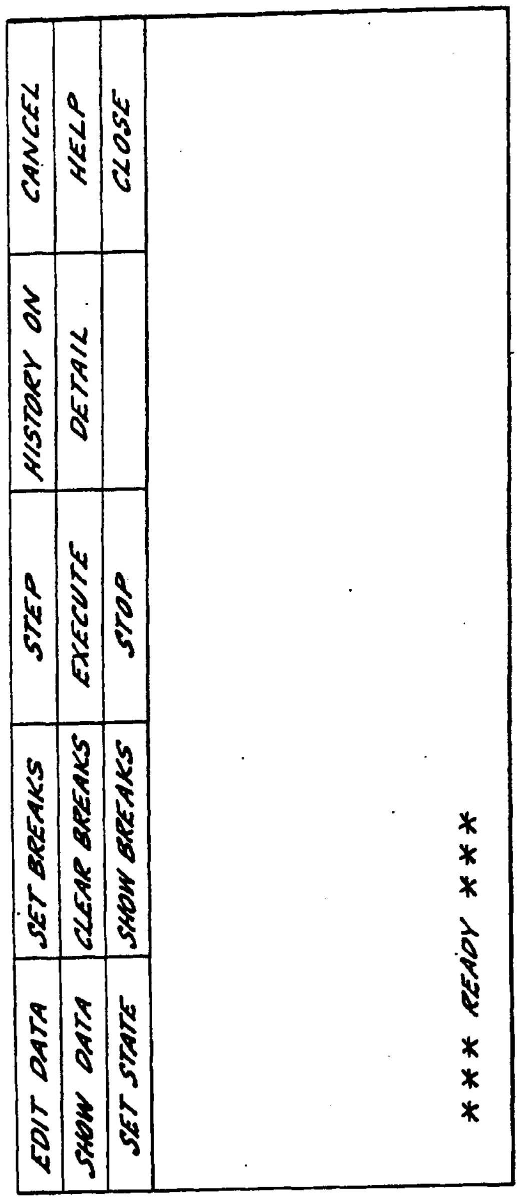

EDIT DATA
SET BREAKS
STEP
HISTORY ON
CANCEL
SHOW DATA
CLEAR BREAKS
EXECUTE
DETAIL
HELP
SET STATE
SHOW BREAKS
STOP
CLOSE
*** READY ***

FIG. 8.

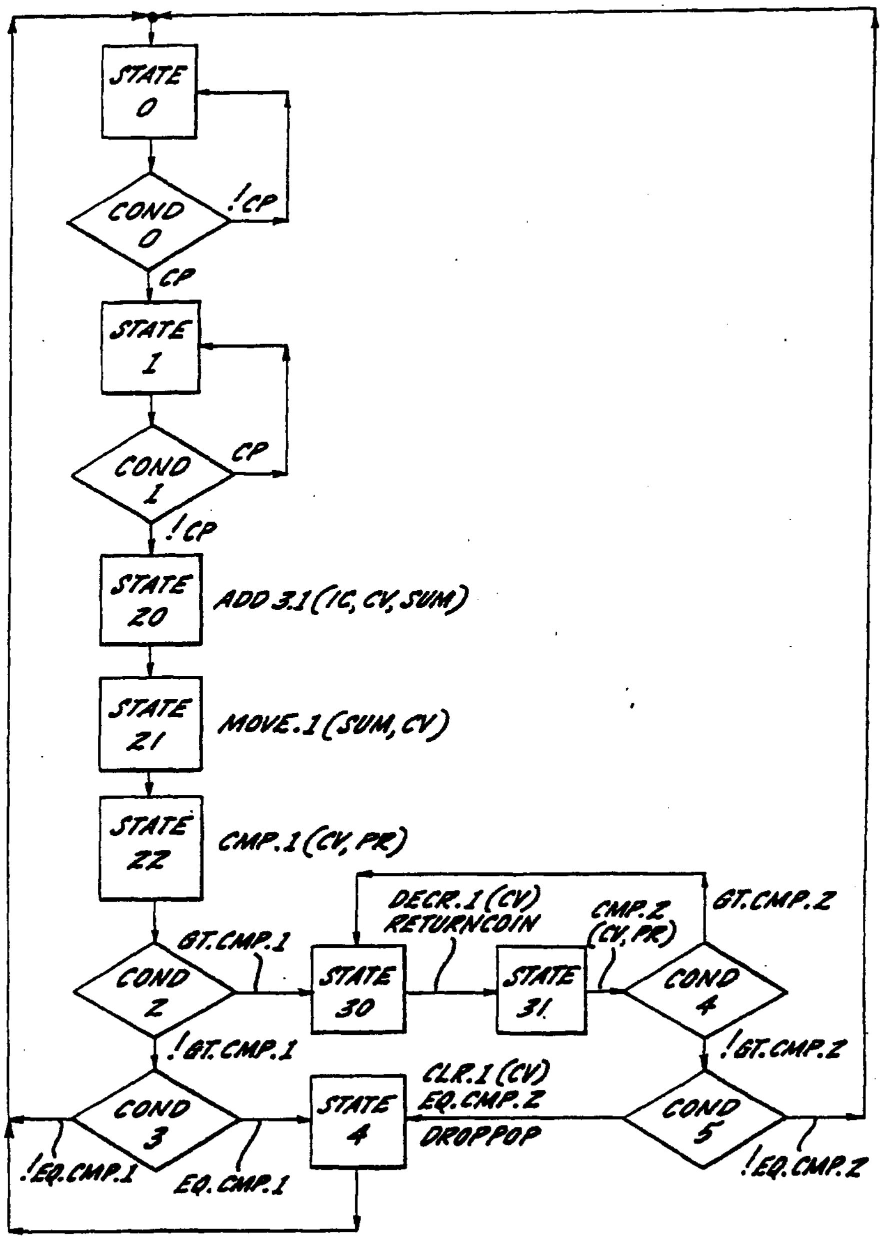

STATE 0
COND 0
!CP
CP
STATE 1
COND 1
CP
!CP
STATE 20
ADD 3.1 (IC, CV, SUM)
STATE 21
MOVE.1 (SUM, CV)
STATE 22
CMP.1 (CV, PR)
COND 2
GT.CMP.1
!GT.CMP.1
STATE 30
DECR.1 (CV)
RETURNCOIN
STATE 31
CMP.2 (CV, PR)
COND 4
GT.CMP.2
!GT.CMP.2
COND 3
!EQ.CMP.1
EQ.CMP.1
STATE 4
CLR.1 (CV)
EQ.CMP.2
DROPPOP
COND 5
!EQ.CMP.2

FIG. 10.

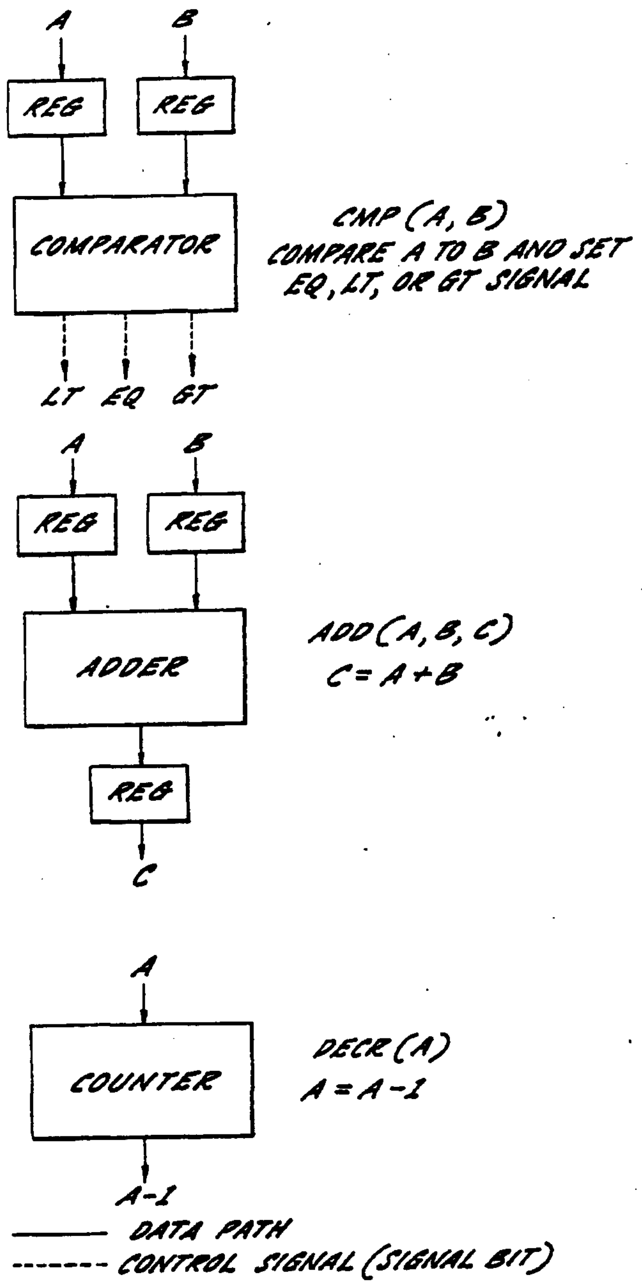

A
B
REG
REG
COMPARATOR
CMP (A, B)
COMPARE A TO B AND SET
EQ, LT, OR GT SIGNAL
LT
EQ
GT
A
B
REG
REG
ADDER
ADD (A, B, C)
C = A + B
REG
C
A
COUNTER
DECR (A)
A = A - 1
A-1
DATA PATH
CONTROL SIGNAL (SIGNAL BIT)

FIG. 11.

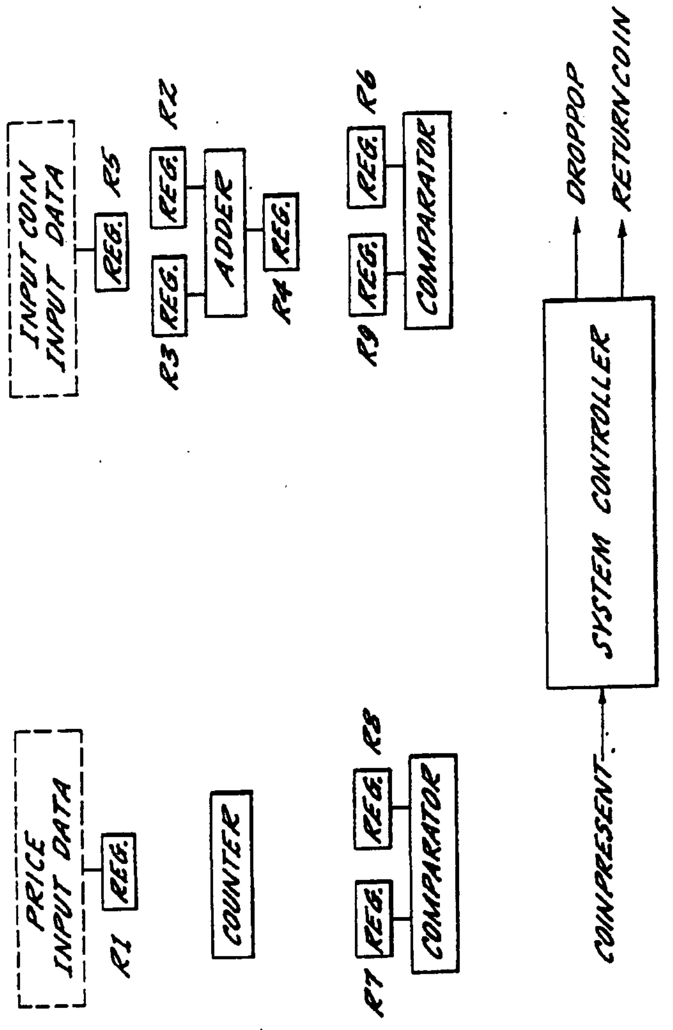

INPUT COIN
INPUT DATA
REG.
R5
R3
REG.
R2
REG.
ADDER
REG.
R4
R9
REG.
R6
REG.
COMPARATOR
DROPPOP
RETURN COIN
SYSTEM CONTROLLER
PRICE
INPUT DATA
REG.
R1
COUNTER
R7
REG.
R8
REG.
COMPARATOR
COINPRESENT

FIG. 12.

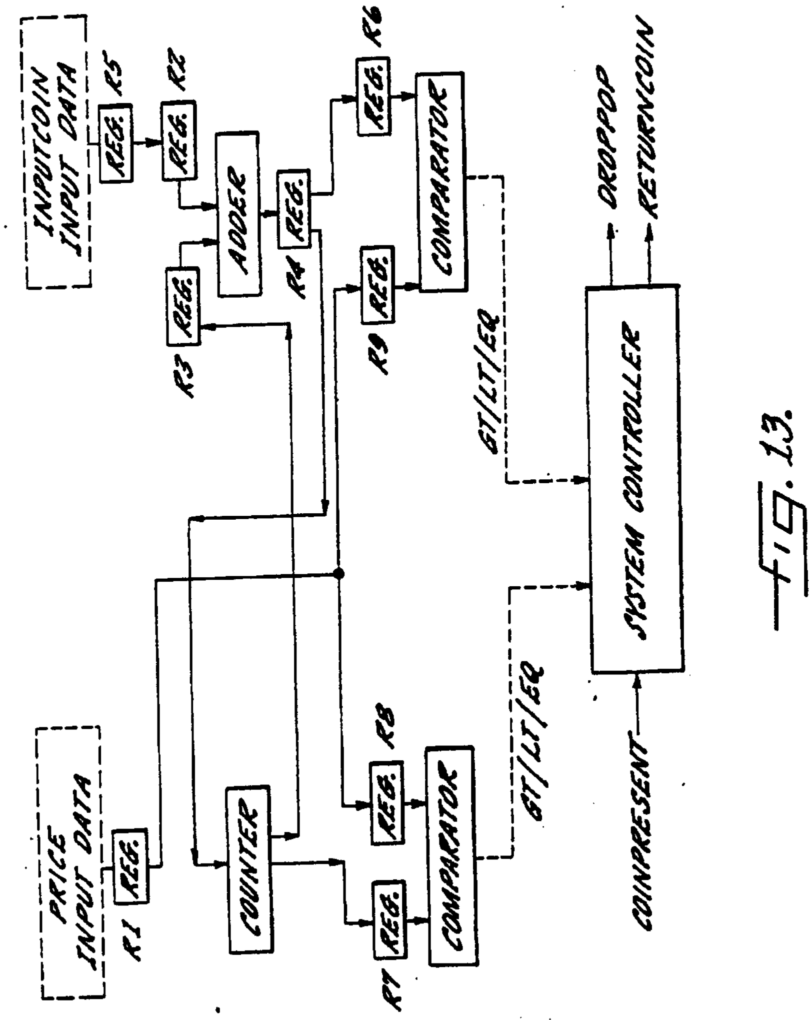

INPUTCOIN INPUT DATA
REG.
R5
REG.
R2
ADDER
REG.
R3
REG.
R4
REG.
R6
REG.
R9
COMPARATOR
GT/LT/EQ
DROPPOP
RETURNCOIN
SYSTEM CONTROLLER
PRICE INPUT DATA
REG.
R1
COUNTER
REG.
R8
REG.
R7
COMPARATOR
GT/LT/EQ
COINPRESENT

FIG. 13.

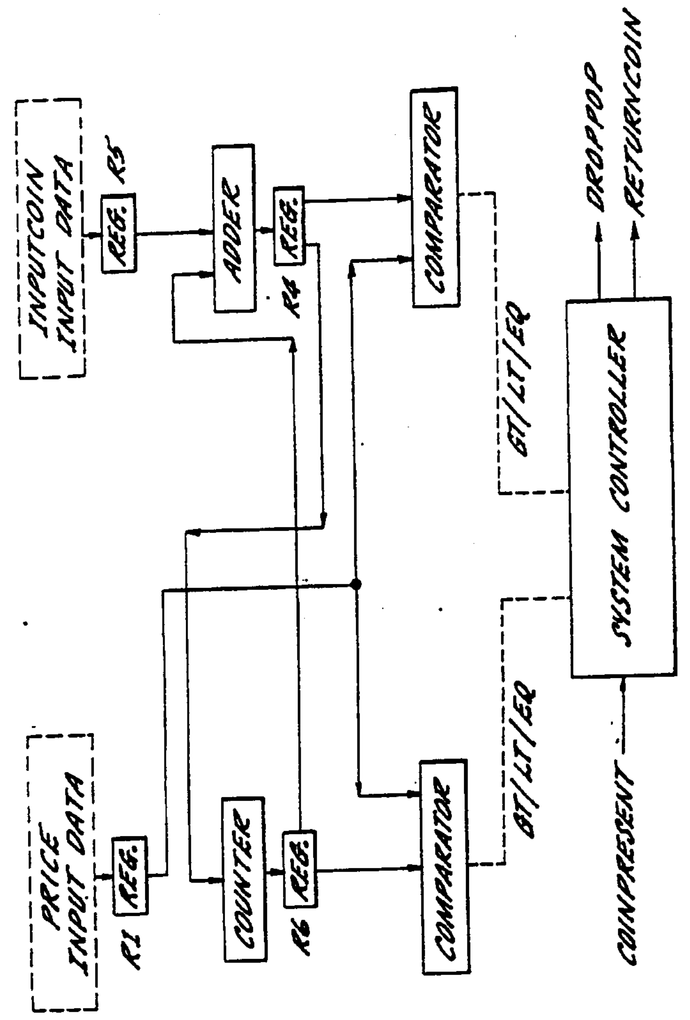

INPUTCOIN INPUT DATA
REG.
R5
ADDER
REG.
R4
COMPARATOR
GT/LT/EQ
DROPPOP
RETURNCOIN
SYSTEM CONTROLLER
PRICE INPUT DATA
REG.
R1
COUNTER
REG.
R6
COMPARATOR
GT/LT/EQ
COINPRESENT

FIG. 14.

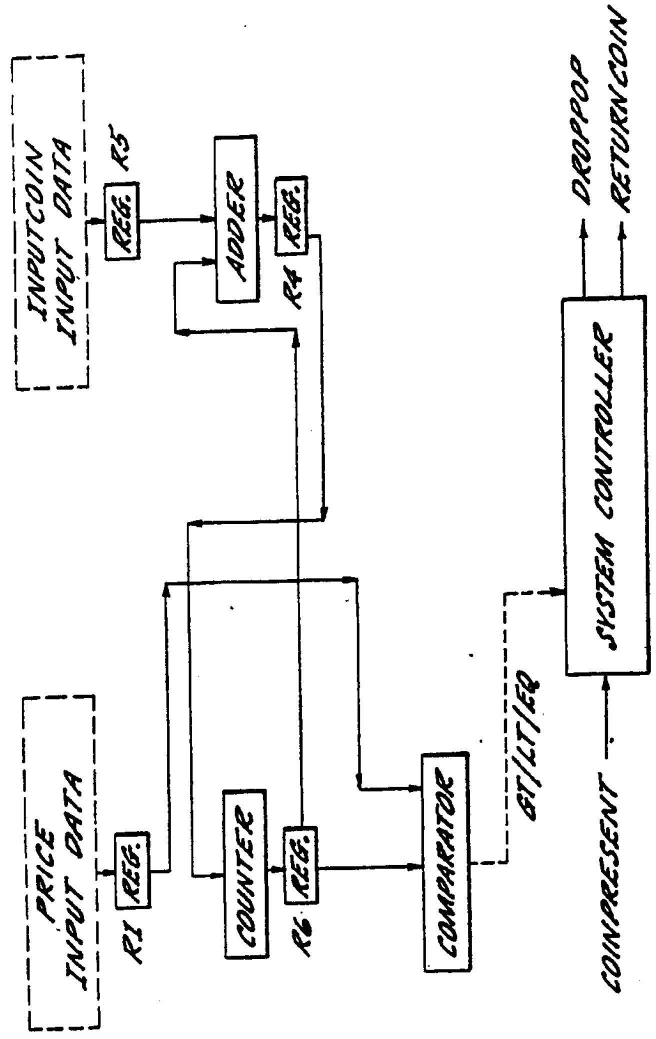

INPUTCOIN INPUT DATA
R5
REG.
ADDER
REG.
R4
DROPPOP
RETURNCOIN
SYSTEM CONTROLLER
PRICE INPUT DATA
REG.
R1
COUNTER
REG.
R6
COMPARATOR
GT/LT/EQ
COINPRESENT

FIG. 15.

KNOWLEDGE BASED METHOD AND APPARATUS FOR DESIGNING INTEGRATED CIRCUITS USING FUNCTIONAL SPECIFICATIONS

FIELD AND BACKGROUND OF THE INVENTION

This invention relates to the design of integrated circuits, and more particularly relates to a computer-aided method and apparatus for designing integrated circuits.

An application specific integrated circuit (ASIC) is an integrated circuit chip designed to perform a specific function, as distinguished from standard, general purpose integrated circuit chips, such as microprocessors, memory chips, etc. A highly skilled design engineer having specialized knowledge in VLSI circuit design is ordinarily required to design a ASIC. In the design process, the VLSI design engineer will consider the particular objectives to be accomplished and tasks to be performed by the integrated circuit and will create structural level design specifications which define the various hardware components required to perform the desired function, as well as the interconnection requirements between these components. A system controller must also be designed for synchronizing the operations of these components. This requires an extensive and all encompassing knowledge of the various hardware components required to achieve the desired objectives, as well as their interconnection requirements, signal level compatibility, timing compatibility, physical layout, etc. At each design step, the designer must do tedious analysis. The design specifications created by the VLSI design engineer may, for example, be in the form of circuit schematics, parameters or specialized hardware description languages (HDLs).

From the structural level design specifications, the description of the hardware components and interconnections is converted to a physical chip layout level description which describes the actual topological characteristics of the integrated circuit chip. This physical chip layout level description provides the mask data needed for fabricating the chip.

Due to the tremendous advances in very large scale integration (VLSI) technology, highly complex circuit systems are being built on a single chip. With their complexity and the demand to design custom chips at a faster rate, in large quantities, and for an ever increasing number of specific applications, computer-aided design (CAD) techniques need to be used. CAD techniques have been used with success in design and verification of integrated circuits, at both the structural level and at the physical layout level. For example, CAD systems have been developed for assisting in converting VLSI structural level descriptions of integrated circuits into the physical layout level topological mask data required for actually producing the chip. Although the presently available computer-aided design systems greatly facilitate the design process, the current practice still requires highly skilled VLSI design engineers to create the necessary structural level hardware descriptions.

There is only a small number of VLSI designers who possess the highly specialized skills needed to create structural level integrated circuit hardware descriptions. Even with the assistance of available VLSI CAD tools, the design process is time consuming and the probability of error is also high because of human involvements. There is a very significant need for a better and more cost effective way to design custom integrated circuits.

SUMMARY OF THE INVENTION

In accordance with the present invention a CAD (computer-aided design) system and method is provided which enables a user to define the functional requirements for a desired target integrated circuit, using an easily understood functional architecture independent level representation, and which generates therefrom the detailed information needed for directly producing an application specific integrated circuit (ASIC) to carry out those specific functions. Thus, the present invention, for the first time, opens the possibility for the design and production of ASICs by designers, engineers and technicians who may not possess the specialized expert knowledge of a highly skilled VLSI design engineer.

The functional architecture independent specifications of the desired ASIC can be defined in a suitable manner, such as in list form or preferably in a flowchart format. The flowchart is a highly effective means of describing a sequence of logical operations, and is well understood by software and hardware designers of varying levels of expertise and training. From the flowchart (or other functional specifications), the system and method of the present invention translates the functional architecture independent specifications into structural an architecture specific level definition of an integrated circuit, which can be used directly to produce the ASIC. The structural level definition includes a list of the integrated circuit hardware cells needed to achieve the functional specifications. These cells are selected from a cell library of previously designed hardware cells of various functions and technical specifications. The system also generates data paths among the selected hardware cells. In addition, the present invention generates a system controller and control paths for the selected integrated circuit hardware cells. The list of hardware cells and their interconnection requirements may be represented in the form of a netlist. From the netlist it is possible using either known manual techniques or existing VLSI CAD layout systems to generate the detailed chip level geometrical information (e.g. mask data) required to produce the particular application specific integrated circuit in chip form.

The preferred embodiment of the system and method of the present invention which is described more fully hereinafter is referred to as a Knowledge Based Silicon Compiler (KBSC). The KBSC is an ASIC design methodology based upon artificial intelligence and expert systems technology. The user interface of KBSC is a flowchart editor which allows the designer to represent VLSI systems in the form of a flowchart. The KBSC utilizes a knowledge based expert system, with a knowledge base extracted from expert ASIC designers with a high level of expertise in VLSI design to generate from the flowchart a netlist which describes the selected hardware cells and their interconnection requirements.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention will be better understood by reference to the detailed description which follows, taken in connection with the accompanying drawings, in which

FIG. 1*a* illustrates a functional level design representation of a portion of a desired target circuit, shown in the form of a flowchart;

FIG. 1*b* illustrates a structural level design representation of an integrated circuit;

FIG. 1*c* illustrates a design representation of a circuit at a physical layout level, such as would be utilized in the fabrication of an integrated circuit chip;

FIG. 2 is a block schematic diagram showing how integrated circuit mask data is created from flowchart descriptions by the KBSC system of the present invention;

FIG. 3 is a somewhat more detailed schematic illustration showing the primary components of the KBSC system;

FIG. 4 is a schematic illustration showing how the ASIC design system of the present invention draws upon selected predefined integrated circuit hardware cells from a cell library;

FIG. 5 is an example flowchart defining a sequence of functional operations to be performed by an integrated circuit;

FIG. 6 is a structural representation showing the hardware blocks and interconnection requirements for the integrated circuit defined in FIG. 5;

FIG. 7 is an illustration of the flowchart editor window;

FIG. 8 is an illustration of the flowchart simulator window;

FIG. 9 is an illustration of the steps involved in cell list generation;

FIG. 10 is an example flowchart for a vending machine system;

FIG. 11 illustrates the hardware components which correspond to each of the three macros used in the flowchart of FIG. 10;

FIG. 12 is an initial block diagram showing the hardware components for an integrated circuit as defined in the flowchart of FIG. 10;

FIG. 13 is a block diagram corresponding to FIG. 12 showing the interconnections between blocks;

FIG. 14 is a block diagram corresponding to FIG. 13 after register optimization; and

FIG. 15 is a block diagram corresponding to FIG. 14 after further optimization.

DETAILED DESCRIPTION OF ILLUSTRATIVE EMBODIMENTS

FIGS. 1*a*, 1*b* and 1*c* illustrate three different levels of representing the design of an integrated circuit. FIG. 1*a* shows a functional (or behavioral) representation architecture independent in the form of a flowchart. A flowchart is a graphic representation of an algorithm and consists of two kinds of blocks or states, namely actions and conditions (decisions). Actions are conventionally represented in the flowchart by a rectangle or box, and conditions are represented by a diamond. Transitions between actions and conditions are represented by lines with arrows. FIG. 1*b* illustrates a structural (or logic) level representation of an integrated circuit. In this representation, blocks are used to represent integrated architecture specific circuit hardware components for performing various functions, and the lines interconnecting the blocks represent paths for the flow of data or control signals between the blocks. The blocks may, for example, represent hardware components such as adders, comparators, registers, system controllers, etc. FIG. 1*c* illustrates a physical layout level representation of an integrated circuit design, which provides the detailed mask data necessary to actually manufacture the devices and conductors which together comprise integrated circuit.

As noted earlier, the design of an integrated circuit at the structural level requires a design engineer with highly specialized skills and expertise in VLSI design. In the KBSC system of the present invention, however, integrated circuits can be designed at a functional level because the expertise in VLSI design is provided and applied by the invention. Allowing the designer to work with flowcharts instead of logic circuit schematics simplifies the task of designing custom integrated circuits, making it quicker, less expensive and more reliable. The designer deals with an algorithm using simple flowcharts at an architecture independent functional (behavioral) level, and needs to know only the necessary logical steps to complete a task, rather than the specific means for accomplishing the task. Designing with flowcharts requires less work in testing because flowcharts allow the designer to work much closer to the algorithm. On the other hand, previously existing VLSI design tools require the designer to represent an algorithm with complex circuit schematics at a structural level, therefore requiring more work in testing. Circuit schematics make it harder for the designer to cope with the algorithm function which needs to be incorporated into the target design because they intermix the hardware and functional considerations. Using flowcharts to design custom integrated circuits will allow a large number of system designers to access VLSI technology, where previously only a small number of designers had the knowledge and skills to create the necessary structural level hardware descriptions.

The overall system flow is illustrated in FIG. 2. The user enters the functional specifications of the circuit into the knowledge based silicon compiler (KBSC) 10 in the form of a flowchart 11. The KBSC 10 then generates a netlist 15 from the flowchart. The netlist 15 includes a custom generated system controller, all other hardware cells required to implement the necessary operations, and interconnection information for connecting the hardware cells and the system controller. The netlist can be used as input to any existing VLSI layout and routing tool 16 to create mask data 18 for geometrical layout.

System Overview

The primary elements or modules which comprise the KBSC system are shown in FIG. 3. In the embodiment illustrated and described herein, these elements or modules are in the form of software programs, although persons skilled in the appropriate art will recognize that these elements can easily be embodied in other forms, such as in hardware.

Referring more particularly to FIG. 3, it will be seen that the KBSC system 10 includes a program 20 called EDSIM, which comprises a flowchart editor 21 for creating and editing flowcharts and a flowchart simulator 22 for simulation and verification of flowcharts. Actions to be performed by each of the rectangles represented in the flowchart are selected from a macro library 23. A program 30 called PSCS (path synthesizer and cell selector) includes a data and control path synthesizer module 31, which is a knowledge based system for data and control path synthesis. PSCS also includes a cell selector 32 for selecting the cells required for system design. The cell selector 32 selects from a cell library 34 of previously designed hardware cells the appropriate cell or cells required to perform each action and condition represented in the flowchart. A controller generator 33 generates a custom designed system controller for controlling the operations of the other hardware cells. The knowledge base 35 contains ASIC design expert knowledge required for data path synthesis and cell selection. Thus, with a functional flowchart input, PSCS generates a system controller, selects all other hardware cells, generates data and control paths, and generates a netlist describing all of this design information.

The KBSC system employs a hierarchal cell selection ASIC design approach, as is illustrated in FIG. 4. Rather than generating every required hardware cell from scratch, the system draws upon a cell library 34 of previously designed, tested and proven hardware cells of various types and of various functional capabilities with a given type. The macro library 23 contains a set of macros defining various actions which can be specified in the flowchart. For each macro function in the macro library 23 there may be several hardware cells in the cell library 34 of differing geometry and characteristics capable of performing the specified function. Using a rule based expert system with a knowledge base 35 extracted from expert ASIC designers, the KBSC system selects from the cell library 34 the optimum cell for carrying out the desired function.

Referring again to FIG. 3, the cells selected by the cell selector 32, the controller information generated by the controller generator 33 and the data and control paths generated by the data/control path synthesizer 31 are all utilized by the PSCS program 30 to generate the netlist 15. The netlist is a list which identifies each block in the circuit and the interconnections between the respective inputs and outputs of each block. The netlist provides all the necessary information required to produce the integrated circuit. Computer-aided design systems for cell placement and routing are commercially available which will receive netlist data as input and will lay out the respective cells in the chip, generate the necessary routing, and produce mask data which can be directly used by a chip foundry in the fabrication of integrated circuits.

System Requirements

The KBSC system can be operated on a suitable programed general purpose digital computer. By way of example, one embodiment of the system is operated in a work station environment such as Sun3 and VAXStation-II/GPX Running UNIX Operating System and X Window Manager. The work station requires a minimum of 8 megabytes of main storage and 20 megabytes of hard disk space. The monitor used is a color screen with 8-bit planes. The software uses C programming language and INGRES relational data base.

The human interface is mainly done by the use of pop up menus, buttons, and a special purpose command language. The permanent data of the integrated circuit design are stored in the data base for easy retrieval and update. Main memory stores the next data temporarily, executable code, design data (flowchart, logic, etc.), data base (cell library), and knowledge base. The CPU performs the main tasks of creating and simulating flowcharts and the automatic synthesis of the design.

Flowchart Example

To describe the mapping of a flowchart to a netlist, consider an example flowchart shown in FIG. 5, which is of part of a larger overall system. In this illustrative flowchart, two variables, VAL1 and VAL2 are compared and if they are equal, they are added together. In this instance, the first action (Action 1) involves moving the value of variable VALI to register A. The second action comprises moving the value of variable VAL2 to register B. Condition 1 comprises comparing the values in registers A and B. Action 3 comprises adding the values of registers A and B and storing the result in register C.

In producing an integrated circuit to carry out the function defined in FIG. 5, the KBSC maps the flowchart description of the behavior of the system to interconnection requirements between hardware cells. The hardware cells are controlled by a system controller which generates all control signals. There are two types of variables involved in a system controller:

(1) Input variables: These are generated by hardware cells, and/or are external input to the controller. These correspond to conditions in the flowchart.

(2) Output variables: These are generated by the system controller and correspond to actions in the flowchart.

FIG. 6 illustrates the results of mapping the flowchart of FIG. 5 onto hardware cells. The actions and the conditions in the flowchart are used for cell selection and data and control path synthesis. The VALI register and VAL2 register and the data paths leading therefrom have already been allocated in actions occurring before Action 1 in our example. Action 1 causes generation of the data register A. Similarly, Action 2 causes the allocation of data register B. The comparator is allocated as a result of the comparison operation in Condition 1. The comparison operation is accomplished by (1) selecting a comparator cell, (2) mapping the inputs of the comparator cell to registers A and B, (3) generating data paths to connect the comparator with the registers A and B and (4) generating input variables corresponding to equal to, greater than, and less than for the system controller. Similarly the add operation in Action 3 causes selection of the adder cell, mapping of the adder parameters to the registers and creating the data paths.

Following this methodology, a block list can be generated for a given flowchart. This block list consists of a system controller and as many other blocks as may be required for performing the necessary operations. The blocks are connected with data paths, and the blocks are controlled by the system controller through control paths. These blocks can be mapped to the cells selected from a cell library to produce a cell list.

Interactive Flowchart Editor and Simulator

The creation and verification of the flowchart is the first step in the VLSI design methodology. The translation from an algorithm to an equivalent flowchart is performed with the Flowchart Editor 21 (FIG. 3). The verification of the edited flowchart is performed by the Flowchart Simulator 22 The Flowchart Editor and Simulator are integrated into one working environment for interactive flowchart editing, with a designer friendly interface.

EDSIM is the program which contains the Flowchart Editor 21 and the Flowchart Simulator 22. It also provides functions such as loading and saving flowcharts. EDSIM will generate an intermediate file, called a statelist, for each flowchart. This file is then used by the PSCS program 30 to generate a netlist.

Flowchart Editor

The Flowchart Editor 21 is a software module used for displaying, creating, and editing the flowchart. This module is controlled through the flowchart editing window illustrated in FIG. 7. Along with editing functions the Flowchart Editor also provides checking of design errors.

The following is a description of the operations of the Flowchart Editor. The main editing functions include, create, edit, and delete states, conditions, and transitions. The create operation allows the designer to add a new state, condition, or transitions to a flowchart. Edit allows the designer to change the position of a state, condition or transition, and delete allows the designer to remove a state, condition or transition from the current flowchart. States which contain actions are represented by boxes, conditions are represented by diamonds, and transitions are represented by lines with arrows showing the direction of the transition.

Edit actions allows the designer to assign actions to each box. These actions are made up of macro names and arguments. An example of arguments is the setting and clearing of external signals. A list of basic macros available in the macro library 23 is shown in Table 1.

TABLE 1

Macro	Description
ADD (A,B,C)	C = A + B
SUB (A,B,C)	C = A − B
MULT (A,B,C)	C = A * B
DIV (A,B,C)	C = A div B
DECR (A)	A = A − 1
INCR (A)	A = A + 1
CLR (A)	A = 0
REG (A,B)	B = A
CMP (A,B)	Compare A to B and set EQ,LT,GT signals
CMP0 (A)	Compare A to 0 and set EQ,LT,GT signals
NEGATE (A)	A = NOT (A)
MOD (A,B,C)	C = A Modulus B
POW (A,B,C)	C = A B
DC2 (A,S1,S2,S3,S4)	Decode A into S1,S2,S3,S4
EC2 (S1,S2,S3,S4,A)	Encode S1,S2,S3,S4 into A
MOVE (A,B)	B = A
CALL sub-flowchart (A,B, . . .)	Call a sub-flowchart. Pass A,B . . .
START (A,B, . . .)	Beginning state of a sub-flowchart
STOP (A,B . . .)	Ending state of a sub-flowchart

The Flowchart Editor also provides a graphical display of the flowchart as the Flowchart Simulator simulates the flowchart. This graphical display consists of boxes, diamonds, and lines as shown in FIG. 7. All are drawn on the screen and look like a traditional flowchart. By displaying the flowchart on the screen during simulation it allows the designer to design and verify the flowchart at the same time.

Flowchart Simulator

The Flowchart Simulator 22 is a software module used for simulating flowcharts. This module is controlled through the simulator window illustrated in FIG. 8. The Flowchart Simulator simulates the transitions between states and conditions in a flowchart. The following is a list of the operations of the Flowchart Simulator:

edit data—Change the value of a register or memory.

set state—Set the next state to be simulated.
set detail or summary display—Display summary or detail information during simulation.
set breaks—Set a breakpoint.
clear breaks—Clear all breakpoints.
show breaks—Display current breakpoints.
step—Step through one transition.
execute—Execute the flowchart.
stop—Stop executing of the flowchart. history ON or history OFF—Set history recording on or off.
cancel—Cancel current operation.
help—Display help screen.
close—Close the simulator window.

The results of the simulation are displayed within the simulator window. Also the editor window will track the flowchart as it is being simulated. This tracking of the flowchart makes it easy to edit the flowchart when an error is found.

Cell Selection

The Cell Selector 32 is a knowledge based system for selecting a set of optimum cells from the cell library 34 to implement a VLSI system. The selection is based on functional descriptions in the flowchart, as specified by the macros assigned to each action represented in the flowchart. The cells selected for implementing a VLSI system depend on factors such as cell function, fabrication technology used, power limitations, time delays etc. The cell selector uses a knowledge base extracted from VLSI design experts to make the cell selection.

To design a VLSI system from a flowchart description of a user application, it is necessary to match the functions in a flowchart with cells from a cell library. This mapping needs the use of artificial intelligence techniques because the cell selection process is complicated and is done on the basis of a number of design parameters and constraints. The concept used for cell selection is analogous to that used in software compilation. In software compilation a number of subroutines are linked from libraries. In the design of VLSI systems, a functional macro can be mapped to library cell.

FIG. 4 illustrates the concept of hierarchical cell selection. The cell selection process is performed in two steps:

(1) selection of functional macros
(2) selection of geometrical cells

A set of basic macros is shown in Table 1. A macro corresponds to an action in the flowchart. As an example, consider the operation of adding A and B and storing the result in C. This function is mapped to the addition macro ADD(X, Y, Z). The flowchart editor and flowchart simulator are used to draw the rectangles, diamonds and lines of the flowchart, to assign a macro selected from the macro library 23 to each action represented in the flowchart, and to verify the functions in flowcharts. The flowchart is converted into an intermediate form (statelist) and input to the Cell Selector.

The Cell Selector uses a rule based expert system to select the appropriate cell or cells to perform each action. If the cell library has a number of cells with different geometries for performing the operation specified by the macro, then an appropriate cell can be selected on the basis of factors such as cell function, process technology used, time delay, power consumption, etc.

The knowledge base of Cell Selector 32 contains information (rules) relating to:

(1) selection of macros
(2) merging two macros

(3) mapping of macros to cells
(4) merging two cells
(5) error diagnostics

The above information is stored in the knowledge base 35 as rules.

Cell List Generation

FIG. 9 shows the cell list generation steps. The first step of cell list generation is the transformation of the flowchart description into a structure that can be used by the Cell Selector. This structure is called the statelist. The blocklist is generated from the statelist by the inference engine. The blocklist contains a list of the functional blocks to be used in the integrated circuit. Rules of the following type are applied during this stage.

map arguments to data paths
map actions to macros
connect these blocks

Rules also provide for optimization and error diagnostics at this level.

The cell selector maps the blocks to cells selected from the cell library 34. It selects an optimum cell for a block. This involves the formulation of rules for selecting appropriate cells from the cell library. Four types of information are stored for each cell. These are:

(1) functional level information: description of the cell at the register transfer level.

(2) logic level information: description in terms of flip-flops and gates.

(3) circuit level information: description at the transistor level.

(4) Layout level information: geometrical mask level specification.

The attributes of a cell are:

cell name
description
function
width
height
status
technology
minimum delay
typical delay
maximum delay
power
file
designer
date
comment
inspector

In the cell selection process, the above information can be used. Some parameters that can be used to map macros to cells are:

(1) name of macro
(2) function to be performed
(3) complexity of the chip
(4) fabrication technology
(5) delay time allowed
(6) power consumption
(7) bit size of macro data paths

Netlist Generation

The netlist is generated after the cells have been selected by PSCS. PSCS also uses the macro definitions for connecting the cell terminals to other cells. PSCS uses the state-to-state transition information from an intermediate form representation of a flowchart (i.e. the

statelist) to generate a netlist. PSCS contains the following knowledge for netlist generation:

(1) Data path synthesis
(2) Data path optimization
(3) Macro definitions
(4) Cell library
(5) Error detection and correction

The above information is stored in the knowledge base 35 as rules. Knowledge engineers help in the formulation of these rules from ASIC design experts. The macro library 23 and the cell library 34 are stored in a database of KBSC.

A number of operations are performed by PSCS. The following is a top level description of PSCS operations:

(1) Read the flowchart intermediate file and build a statelist.

(2) current_context=START

(3) Start the inference engine and load the current context rules.

(4) Perform one of the following operations depending upon current_context:

(a) Modify the statelist for correct implementation.
(b) Create blocklist, macrolist and data paths.
(c) Optimize blocklist and datapath list and perform error checks.
(d) Convert blocks to cells.
(e) Optimize cell list and perform error checks.
(f) Generate netlist.
(g) Optimize netlist and perform error checks and upon completion Goto 7.

(5) If current_context has changed, load new context rules.

(6) Goto 4.

(7) Output netlist file and stf files and Stop.

In the following sections, operations mentioned in step 4 are described. The Rule Language and PSCS display are also described.

Rule Language

The rule language of PSCS is designed to be declarative and to facilitate rule editing. In order to make the expert understand the structure of the knowledge base, the rule language provides means for knowledge representation. This will enable the format of data structures to be stated in the rule base, which will enable the expert to refer to them and understand the various structures used by the system. For example, the expert can analyze the structure of wire and determine its components. The expert can then refer these components into rules. If a new object has to be defined, then the expert can declare a new structure and modify some existing structure to link to this new structure. In this way, the growth of the data structures can be visualized better by the expert. This in turn helps the designer to update and append rules.

The following features are included in the rule language:

(i) Knowledge representation in the form of a record structure.

(ii) Conditional expressions in the antecedent of a rule.

(iii) Facility to create and destroy structure in rule actions.

(iv) The assignment statement in the action of a rule.

(v) Facility for input and output in rule actions.

(vi) Provide facility to invoke C functions from rule actions.

The rule format to be used is as follows:

DEF012581

The rule format to be used is as follows:

```
Rule    <number> <context>
If {
        <if-clause>
}
Then    {
        <then-clause>
}
where   <number>          rule number
        <context>         context in which this rule is
                          active
        <if-clause>       the condition part of the rule
        <then-clause>     the action part of the rule
```

Inference Strategy

The inference strategy is based on a fast pattern matching algorithm. The rules are stored in a network and the requirement to iterate through the rules is avoided. This speeds up the execution. The conflict resolution strategy to be used is based on the following:

(1) The rule containing the most recent data is selected.

(2) The rule which has the most complex condition is selected.

(3) The rule declared first is selected.

Rule Editor

PSCS provides an interactive rule editor which enables the expert to update the rule set. The rules are stored in a database so that editing capabilities of the database package can be used for rule editing. To perform this operation the expert needs to be familiar with the various knowledge structures and the inferencing process. If this is not possible, then the help of a knowledge engineer is needed.

PSCS provides a menu from which various options can be set. Mechanisms are provided for setting various debugging flags and display options, and for the overall control of PSCS.

Facility is provided to save and display the blocklist created by the user. The blocklist configuration created by the user can be saved in a file and later be printed with a plotter. Also the PSCS display can be reset to restart the display process.

PSCS Example Rules:		
Rule 1		
	IF	no blocks exist
	THEN	generate a system controller.
Rule 2		
	IF	a state exists which has a macro AND this macro has not been mapped to a block
	THEN	find a corresponding macro in the library and generate a block for this macro.
Rule 3		
	IF	there is a transition between two states AND there are macros in these states using the same argument
	THEN	make a connection from a register corresponding to the first macro to another register corresponding to the second macro.
Rule 4		
	IF	a register has only a single connection from another register
	THEN	combine these registers into a single register.
Rule 5		
	IF	there are two comparators AND input data widths are of the same size AND one input of these is same AND the outputs of the comparators are used to perform the same operation.
	THEN	combine these comparators into a single comparator.
Rule 6		
	IF	there is a data without a register
	THEN	allocate a register for this data.
Rule 7		
	IF	all the blocks have been interconnected AND a block has a few terminals not connected
	THEN	remove the block and its terminals, or issue an error message.
Rule 8		
	IF	memory is to be used, but a block has not been created for it
	THEN	create a memory block with data, address, read and write data and control terminals.
Rule 9		
	IF	a register has a single connection to a counter
	THEN	combine the register and the counter; remove the register and its terminals.
Rule 10		
	IF	there are connections to a terminal of a block from many different blocks
	THEN	insert a multiplexor; remove the connections to the terminals and connect them to the input of the multiplexor; connect the output of the multiplexor to the input of the block.

Additional rules address the following points:
remove cell(s) that can be replaced by using the outputs of other cell(s)
reduce multiplexor trees
use fan-out from the cells, etc.

Soft Drink Vending Machine Controller Design Example

The following example illustrates how the previously described features of the present invention are employed in the design of an application specific integrated circuit (ASIC). In this illustrative example the ASIC is designed for use as a vending machine controller. The vending machine controller receives a signal each time a coin has been deposited in a coin receiver. The coin value is recorded and when coins totalling the correct amount are received, the controller generates a signal to dispense a soft drink. When coins totalling more than the cost of the soft drink are received, the controller dispenses change in the correct amount.

This vending machine controller example is patterned after a textbook example used in teaching digital system controller design. See Fletcher, William I., *An Engineering Approach to Digital Design*, Prentice-Hall, Inc., pp. 491-505. Reference may be made to this textbook example for a more complete explanation of this vending machine controller requirements, and for an understanding and appreciation of the complex design procedures prior to the present invention for designing the hardware components for a controller.

FIG. 10 illustrates a flowchart for the vending machine controller system. This flowchart would be entered into the KBSC system by the user through the flowchart editor. Briefly reviewing the flowchart, the controller receives a coin present signal when a coin is received in the coin receiver. State0 and cond0 define a waiting state awaiting deposit of a coin. The symbol CP represents "coin present" and the symbol !CP represents "coin not present". State1 and cond1 determine when the coin has cleared the coin receiver. At state20, after receipt of a coin, the macro instruction ADD3.1 (lc, cv, sum) instructs the system to add lc (last coin) and cv (coin value) and store the result as sum. The macro instruction associated with state21 moves the value in the register sum to cv. The macro CMP.1 at state22 compares the value of cv with PR (price of soft drink) and returns signals EQ, GT and LT. The condition cond2 tests the result of the compare operation CMP.1. If the result is "not greater than" (!GT.CMP.1), then the condition cond3 tests to see whether the result is "equal" (EQ.CMP.1). If the result is "not equal" (!EQ.CMP.1), then control is returned to state0 awaiting the deposit of another coin. If cond3 is EQ, then state4 generates a control signal to dispense a soft drink (droppop) and the macro instruction CLR.1(cv) resets cv to zero awaiting another customer.

If the total coins deposited exceed the price, then state30 produces the action "returncoin". Additionally, the macro DECR.1 (cv) reduces the value of cv by the amount of the returned coin. At state31 cv and PR are again compared. If cv is still greater than PR, then control passes to state30 for return of another coin. The condition cond5 tests whether the result of CMP.2 is EQ and will result in either dispensing a drink (droppop) true or branching to state0 awaiting deposit of another coin. The macros associated with the states shown in FIG. 10 correspond to those defined in Table 1 above and define the particular actions which are to be performed at the respective states.

Appendix A shows the intermediate file or "statelist" produced from the flowchart of FIG. 10. This statelist is produced as output from the EDSIM program 20 and is used as input to the PSCS program 30 (FIG. 3).

FIG. 11 illustrates for each of the macros used in the flowchart of FIG. 10, the corresponding hardware blocks. It will be seen that the comparison macro CMP (A,B) results in the generation of a register for storing value A, a register for storing value B, and a comparator block and also produces control paths to the system controller for the EQ, LT, and GT signals generated as a result of the comparison operation. The addition macro ADD (A,B,C) results in the generation of a register for each of the input values A and B, a register for the output value C, and in the generation of an adder block. The macro DECR (A) results in the generation of a counter block. The PSCS program 30 maps each of the macros used in the flowchart of FIG. 10 to the corresponding hardware components results in the generation of the hardware blocks shown in FIG. 12. In generating the illustrated blocks, the PSCS program 30 relied upon rules 1 and 2 of the above listed example rules.

FIG. 13 illustrates the interconnection of the block of FIG. 12 with data paths and control paths. Rule 3 was used by the data/control path synthesizer program 31 in mapping the data and control paths.

FIG. 14 shows the result of optimizing the circuit by applying rule 4 to eliminate redundant registers. As a result of application of this rule, the registers R2, R3, R7, R8, and R9 in FIG. 13 were removed. FIG. 15 shows the block diagram after further optimization in which redundant comparators are consolidated. This optimization is achieved in the PSCS program 30 by application of rule 5.

Having now defined the system controller block, the other necessary hardware blocks and the data and control paths for the integrated circuit, the PSCS program 30 now generates a netlist 15 defining these hardware components and their interconnection requirements. From this netlist the mask data for producing the integrated circuit can be directly produced using available VLSI CAD tools.

```
name rpop;
data path @lc<0:5>, cv<0:5>, sum<0:5>, @pr<0:5>;
{
state4 : state0;
state30 : state31;
state21 : state22;
state20 : state21;
state0 :. !cp state0;
state0 :. cp state1;
state1 :. cp state1;
state1 :. !cp state20;
state22 :. GT.CMP.1 state30;
state22 :. !GT.CMP.1*EQ.CMP.1 state4;
state22 :. !GT.CMP.1*!EQ.CMP.1 state0;
state31 :. GT.CMP.2 state30;
state31 :. !GT.CMP.2*EQ.CMP.2 state4;
state31 :. !GT.CMP.2*!EQ.CMP.2 state0;
state30 :: returncoin;
state30 :: DECR.1(cv);
state4 :: droppop;
state4 :: CLR.1(cv);
state31 :: CMP.2(cv,pr);
state22 :: CMP.1(cv,pr);
state21 :: MOVE.1(sum,cv);
state20 :: ADD3.1(lc,cv,sum);
}
```

That which I claimed is:

1. A computer-aided design system for designing an application specific integrated circuit directly from architecture independent functional specifications for the integrated circuit, comprising

a macro library defining a set of architecture independent operations comprised of actions and conditions;

input specification means operable by a user for defining architecture independent functional specifications for the integrated circuit, said functional specifications being comprised of a series of operations comprised of actions and conditions, said input specification means including means to permit the user to specify for each operation a macro selected from said macro library;

a cell library defining a set of available integrated circuit hardware cells for performing the available operations defined in said macro library;

cell selection means for selecting from said cell library for each macro specified by said input specification means, appropriate hardware cells for performing the operation defined by the specified macro, said cell selection means comprising an expert system including a knowledge base containing rules for. selecting hardware cells from said cell library and inference engine means for selecting appropriate hardware cells from said cell library in accordance with the rules of said knowledge base; and

netlist generator means cooperating with said cell selection means for generating as output from the system a netlist defining the hardware cells which are needed to achieve the functional requirements of the integrated circuit and the connections therebetween.

2. The system as defined in claim 1 wherein said input means comprises means specification for receiving user

input of a list defining the series of actions and conditions.

3. The system as defined in claim 1 additionally including mask data generator means for generating from said netlist the mask data required to produce an integrated circuit having the specified functional requirements.

4. The system as defined in claim 1 wherein said input means comprises flowchart editor means specification for creating a flowchart having elements representing said series of actions and conditions.

5. The system as defined in claim 4 additionally including flowchart simulator means for simulating the functions defined in the flowchart to enable the user to verify the operation of the integrated circuit.

6. The system as defined in claim 1 additionally including data path generator means cooperating with said cell selection means for generating data paths for the hardware cells selected by said cell selection means.

7. The system as defined in claim 6 wherein said data path generator means comprises a knowledge base containing rules for selecting data paths between hardware cells and inference engine means for selecting data paths between the hardware cells selected by said cell selection means in accordance with the rules of said knowledge base and the arguments of the specified macros.

8. The system as defined in claim 6 additionally including control generator means for generating a controller and control paths for the hardware cells selected by said cell selection means.

9. A computer-aided design system for designing an application specific integrated circuit directly from a flowchart defining architecture independent functional requirements of the integrated circuit comprising

a marco library defining a set of architecture independent operations comprised of actions and conditions;

flowchart editor means operable by a user for creating a flowchart having elements representing said architecture independent operations;

said flowchart editor means including macro specification means for permitting the user to specify for each operation represented in the flowchart a macro selected from said macro library;

a cell library defining a set of available integrated circuit hardware cells for performing the available operations defined in said macro library;

cell selection means for selecting form said cell library for each specified macro, appropriate hardware cells for performing the operation defined by the specified macro, said cell selection means comprising an expert system including a knowledge base containing rules for selecting hardware cells from said cell library and inference engine means for selecting appropriate hardware cells from said cell library in accordance with the rules of said knowledge base; and

data path generator means cooperating with said cell selection means for generating data paths for the hardware cells selected by said cell selector means, said data path generator means comprising a knowledge base containing rules for selecting data paths between hardware cells and inference engine means for selecting data paths between hardware cells selected by said cell selection means in accordance with the rules of said knowledge base and the arguments of the specified macros.

10. The system as defined in claim 9 additionally including control generator means for generating a controller and control paths for the hardware cells selected by said cell selection means.

11. A computer-aided design system for designing an application specific integrated circuit directly from a flowchart defining architecture independent functional requirements of the integrated circuit, comprising

flowchart editor means operable by a user for creating a flowchart having boxes representing architecture independent actions, diamonds representing architecture independent conditions, and lines with arrows representing transitions between actions and condition and including means for specifying for each box or diamond, a particular action or condition to be performed;

a cell library defining a set of available integrated circuit hardware cells for performing actions and conditions;

a knowledge base containing rules for selecting hardware cells from said cell library and for generating data and control paths for hardware cells; and

expert system means operable with said knowledge base for translating the flowchart defined by said flowchart editor means into a netlist defining the necessary hardware cells and data and control paths required in an integrated circuit having the specified functional requirements.

12. The system as defined in claim 11 including mask data generator means for generating from said netlist the mask data required to produce an integrated circuit having the specified functional requirements.

13. A computer-aided design process for designing an application specific integrated circuit which will perform a desired function comprising

storing a set of definitions of architecture independent actions and conditions;

storing data describing a set of available integrated circuit hardware cells for performing the actions and conditions defined in the stored set;

storing in an expert system knowledge base a set of rules for selecting hardware cells to perform the actions and conditions;

describing for a proposed application specific integrated circuit a series of architecture independent actions and conditions;

specifying for each described action and condition of the series one of said stored definitions which corresponds to the desired action or condition to be performed; and

selecting from said stored data for each of the specified definitions a corresponding integrated circuit hardware cell for performing the desired function of the application specific integrated circuit, said step of selecting a hardware cell comprising applying to the specified definition of the action or condition to be performed, a set of cell selection rules stored in said expert system knowledge base and generating for the selected integrated circuit hardware cells, a netlist defining the hardware cells which are needed to perform the desired function of the integrated circuit and the interconnection requirements therefor.

14. A process as defined in claim 13, including generating from the netlist the mask data required to produce an integrated circuit having the desired function.

15. A process as defined in claim 13 including the further step of generating data paths for the selected integrated circuit hardware cells.

16. A process as defined in claim 15 wherein said step of generating data paths comprises applying to the selected cells a set of data path rules stored in a knowledge base and generating the data paths therefrom.

17. A process as defined in claim 16 including the further step of generating control paths for the selected integrated circuit hardware cells.

18. A knowledge based design process for designing an application specific integrated circuit which will perform a desired function comprising

storing in a macro library a set of macros defining architecture independent actions and conditions;

storing in a cell library a set of available integrated circuit hardware cells for performing the actions and conditions;

storing in a knowledge base set of rules for selecting hardware cells from said cell library to perform the actions and conditions defined by the stored macros;

describing for a proposed application specific integrated circuit a flowchart comprised of elements representing a series of architecture independent actions and conditions which carry out the function to be performed by the integrated circuit;

specifying for each described action and condition of said series a macro selected from the macro library which corresponds to the action or condition; and

applying rules of said knowledge base to the specified macros to select from said cell library the hardware cells required for performing the desired function of the application specific integrated circuit and generating for the selected integrated circuit hardware cells, a netlist defining the hardware cells which are needed to perform the desired function of the integrated circuit and the interconnection requirements therefor.

19. A process as defined in claim 18 also including the steps of

storing in said knowledge base a set of rules for creating data paths between hardware cells, and

applying rules of said knowledge base to the specified means to create data paths for the selected hardware cells.

20. A process as defined in claim 19 also including the steps of generating a controller and generating control paths for the selected hardware cells.

* * * * *

INTERNATIONAL JOURNAL OF COMPUTER AIDED VLSI DESIGN

VOLUME 1
NUMBER 4

SPECIAL ISSUE: PART 2
VLSI CAD IN JAPAN

EDITOR
GEORGE W. ZOBRIST

ASSOCIATE EDITORS
ED CERNY
HIDEAKI KOBAYASHI
PAOLO PRINETTO
HARRY TYRER

ABLEX PUBLISHING CORPORATION
NORWOOD, NEW JERSEY

International Journal of Computer Aided VLSI Design

Volume 1, Number 4, 1989

SPECIAL ISSUE: Part II

VLSI CAD IN JAPAN

Hideaki Kobayashi, Guest Editor

CONTENTS

INTERNATIONAL JOURNAL OF COMPUTER AIDED VLSI DESIGN

EDITOR
George W. Zobrist
University of Missouri-Rolla

ASSOCIATE EDITORS
Ed Cerny
University of Montreal, Canada
Hideaki Kobayashi
University of South Carolina
Paolo Prinetto
Politecnico di Torino, Italy
Harry Tyrer
University of Missouri-Columbia

EDITORIAL BOARD
Magdy A. Bayoumi
University of Southwestern Louisiana
Joe Charlson
University of Missouri-Columbia
Salim Chowdhury
University of Iowa
Sunil R. Das
University of Ottawa, Canada
Tao Li
University of Monash, Australia
R. E. Massara
University of Essex, U.K.
L. M. Patnaik
Indian Institute of Science, Bangalore, India
Sadiq Sait
University of Petroleum/Minerals, Saudi Arabia
Kewal Saluja
University of Wisconsin-Madison
Majid Sarrafzadeh
Northwestern University, Illinois
Steven P. Smith
Microelectronic and Computer Technology Corp., Austin, Texas
Mehmet Yanilmaz
Northwestern University

International Journal of Computer Aided VLSI Design is published quarterly by Ablex Publishing Corporation, 355 Chestnut Street, Norwood, New Jersey 07648

Volume 1, 1989

Individual Subscription: $32.50

Institutional Subscription: $85.00

An additional charge of $12.00 will be made for overseas postage.

Copyright © 1989. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, microfilming, recording, or otherwise, without permission of the publisher.

Printed in U.S.A. ISSN 1042-7988

International Journal of Computer Aided VLSI Design

Volume 1, Number 4, 1989

SPECIAL ISSUE: Part II

VLSI CAD IN JAPAN

Hideaki Kobayashi, Guest Editor

CONTENTS

International Journal of Computer Aided VLSI Design 1, 351–355 (1989)

Part II
Guest Editorial

HIDEAKI KOBAYASHI

Application-specific integrated circuit (ASIC) users in Japan are mainly "set makers" (system designers) who are not experts in VLSI design. They use (TTL) transistor-transistor logic-based techniques to design logic circuits (or netlists). Test vectors often are not generated by ASIC users. Many circuits designed by ASIC users do not implement initial chip functions. This often results in problems associated with design responsibilities between ASIC users and semiconductor makers.

With advanced CAD tools and systems, the interface between ASIC users and semiconductor makers will be shifted to a higher functional level. ASIC users can enter desired chip functions using optimum input forms such as truth tables, state diagrams, flowcharts, and hardware description languages (HDLs). ASIC users can also ensure initial chip functions without generating test vectors at a structural (circuit) level. Chips can be developed by ASIC users who are not familiar with circuit design or semiconductor technology.

Automatic translation between behavioral or functional inputs to logic circuits is provided by logic synthesis systems. There are two different types of approaches to automatic logic synthesis: algorithmic and rule-based. IF–THEN-type rules rather than algorithmic programming languages are used in the latter approach to synthesize logic circuits. These rules are extracted from expert ASIC designers who have been designing chips for many years.

Many Japanese semiconductor makers have developed in-house software for logic synthesis. A variety of commercial software for logic synthesis is also available in the Japanese market. Table 1 shows an example list of commercial software for logic synthesis. Table 2 shows an example list of in-house software for logic synthesis. These tables are translated from the article entitled "Logic Synthesis Software for ASIC Makes Design at the Functional Level Possible" by U. Kojima, published in *Nikkei Electronics*, November 1988 issue. Target applications of ASICs designed by these logic synthesis systems include audio visual machines, communications large-scale integration (LSI), digital signal processors, general-purpose microprocessors, micro program control, office automation, and peripheral LSI.

A list of other logic synthesis systems is provided in Table 3. These systems perform synthesis by translating a functional (algorithmic) description into a structural (data path and controller) description. They are compared in the second article of this special issue.

Table 1. Commercial Logic Synthesis Software

Name:	ZEPHCAD
Vendor:	Fujitsu Ltd.
Input form:	State diagram, Boolean expression, truth table
Design style:	CMOS gate array, CMOS standard cell
Name:	PARTHENON
Vendor:	NTT Data Communications Systems Corporation
Input form:	HDL (SFL)
Design style:	CMOS gate array, CMOS standard cell
Name:	Logic Synthesizer
Vendor:	LSI Logic Corporation
Input form:	State diagram, Truth table, Boolean expression, parameter
Design style:	CMOS gate array, CMOS standard cell
Name:	Finesse
Vendor:	Seattle Silicon Corporation
Input form:	Truth table, Boolean expression, state diagram
Design style:	CMOS standard cell
Name:	Logic Compiler
Vendor:	Silicon Compiler Systems Corporation
Input form:	Boolean expression, truth table, functional diagram, state diagram, HDL
Design style:	CMOS standard cell
Name:	SilcSyn
Vendor:	Silc Technologies, Inc.
Input form:	HDL (DDL)
Design style:	CMOS gate array, CMOS standard cell
Name:	Logic Consultant
Vendor:	Trimeter Technologies Corporation
Input form:	Boolean expression, netlist
Design style:	CMOS gate array
Name:	Design Compiler
Vendor:	Synopsys, Inc.
Input form:	HDL (Verilog), Boolean expression, truth table, netlist
Design style:	CMOS gate array, CMOS standard cell
Name:	State Machine Compiler
Vendor:	VLSI Technology, Inc.
Input form:	Boolean expression, truth table, parameter, state diagram, netlist, functional diagram, HDL (VHDL)
Design style:	CMOS gate array, CMOS standard cell

Table 2. In-House Logic Synthesis Software

Name:	FLORA
Company:	Fujitsu, Ltd.
Input form:	Functional diagram
Results:	Over 100 chips
Name:	ProLogic
Company:	Hitachi, Ltd.
Input form:	Structural and behavioral description of datapath and control circuits
Results:	Several samples under evaluation
Name:	LODES
Company:	Matsushita Electric Industrial Co., Ltd.
Input form:	Functional diagram, Boolean expression, truth table, HDL (HSL-FX)
Results:	Two chips for practical use
Name:	TL/C
Company:	Matsushita Electric Industrial Co., Ltd.
Input form:	Circuit diagram, Boolean expression, truth table, state diagram
Results:	Evaluation completed
Name:	LORES/EX
Company:	Mitsubishi Electric Corp.
Input form:	Functional diagram
Results:	Under evaluation
Name:	Fusion
Company:	NEC Corp.
Input form:	HDL (FDL)
Results:	Two chips
Name:	EXLOG
Company:	NEC Corp.
Input form:	HDL (FDL)
Results:	One chip
Name:	ANGEL
Company:	NEC Corp.
Input form:	HDL (HSL-FX)
Results:	Two chips
Name:	CLS
Company:	Oki Electric Industry Co., Ltd.
Input form:	HDL (HSL-FX)
Applications:	Microprocessors, DSP, control LSI
Results:	Five chips
Name:	Logic Synthesis System
Company:	Sharp Corp.
Input form:	Boolean expression, truth table, functional diagram
Results:	Under evaluation
Name:	Logic Synthesis Systems
Company:	Toshiba Corp.
Input form:	HDL (HHDL)
Results:	Thirty chips

Table 3. Other Logic Synthesis Software

Name:	CMU-DA System
Company:	Carnegie-Mellon University
Input form:	IPS language
Results:	Not available
Name:	VLSI Design Automation Assistant
Company:	AT&T Bell Laboratories
Input form:	ISPS language
Results:	Several digital systems
Name:	ALERT
Company:	IBM Watson Research Center
Input form:	Iverson's APL
Results:	Not available
Name:	Flamel
Company:	Stanford University
Input form:	Pascal
Results:	Not available
Name:	KBSC
Company:	Ricoh–International Chip Corporation
Input form:	Flowchart
Results:	Ten chips

The first article is "Unified Hardware Description Language and Its Support Tools" from Fujitsu Laboratories Ltd. A new hardware description language called UHDL (Unified Hardware Description Language) is proposed to describe asynchronous behavior and to synthesize large circuits. A hierarchical structure can be described by UHDL from several viewpoints, such as behavior, interface, and data paths.

The second article is "KBSC: A Knowledge-Based Approach to Automatic Logic Synthesis" from International Chip Corporation and Ricoh Company. A knowledge-based silicon compiler (KBSC) provides users with a front-end graphic interface to automatic logic synthesis. Both data-path and control circuits are synthesized by using rules extracted from expert ASIC designers. KBSC's flowchart input form allows ASIC users to enter desired chip functions easily, and verify these functions to ensure functional correctness.

The third article is "Development and Evaluation of an Architecture Design System for Application-Specific Integrated Circuits" from Waseda University. In this article, automatic architecture synthesis for special-purpose integrated circuits is discussed. The authors suggest that their system can synthesize and modify an initial circuit from a given algorithm. Target architectures are small- to medium-scale integrated circuits and are used to implement algorithms for digital signal processing.

The precise analysis of submicron pattern imaging becomes more important as feature sizes decrease. The minimum feature size of a 16-Mbit dynamic random-access memory is approximately half-microns. The fourth article is "A Novel Fast Calculation Method for Partial Coherent Images with Optical Aberration and Its Application to Submicron Pattern Imaging" from

Hitachi Ltd. This method calculates images of isolated square and isolated elbow mask patterns with various optical aberrations. It also calculates image distortions with several types of aberrations simultaneously.

Hideaki Kobayashi Dr. Kobayashi received both M.S. and Ph.D. degrees in Electrical Engineering from Waseda University, Tokyo, Japan. He joined the Department of Electrical and Computer Engineering at the University of South Carolina in 1980, where he is currently an Associate Professor. He is also Director of the VLSI/AI Design Laboratory at the Center for Machine Intelligence located at the new Swearingen Engineering Center, which is one of the most advanced facilities of its kind in the world. His current research interests include VLSI architecture for AI applications and knowledge-based silicon compilation. In addition to teaching on campus, his VLSI courses also are taught throughout the United States via the National Technological University.

International Chip Corporation was established in 1985 in Columbia, South Carolina as culmination of over ten years of research by Dr. Kobayashi, one of the two principals in the firm who also serves as Chairman of the Board. Since 1986 the company has funded continuing research at the University of South Carolina in a private-public partnership with Carolina Research and Development Foundation. For-profit business applications and related product development are conducted at International Chip Corporation's corporate headquarters, within walking distance of the main campus of the University of South Carolina.

Dr. Kobayashi has published over 50 professional papers in leading United States, Japanese, and European journals. He serves as an Associate Editor of the International Journal of Computer Aided VLSI Design. He is frequently invited to speak at professional meetings and seminars relating to ASIC (application specific integrated circuit) design, VLSI CAD, knowledge-based systems, and other associated topics. He also has been the recipient of research grants from several sources, including the National Science Foundation and the Semiconductor Research Corporation. His career contributions have been recognized in several ways, including selection as one of the Outstanding Young Men of America for 1985. He is a member of IEEE and Eta Kappa Nu.

International Journal of Computer Aided VLSI Design 1, 377–390 (1989)

KBSC: A Knowledge-Based Approach to Automatic Logic Synthesis

HIDEAKI KOBAYASHI
International Chip Corporation

TERUMI SUEHIRO AND MASAHIRO SHINDO
Ricoh Company, Ltd.

A knowledge-based silicon compiler (KBSC) has been jointly developed by Ricoh Company and International Chip Corporation. KBSC provides designers with a front-end graphic interface to automatic logic synthesis. A rule-based expert system synthesizes both data-path and control circuits. KBSC automatically translates an algorithmic description in flowchart form into a logic circuit (netlist) that consists of previously registered cells. ASIC design by KBSC is compared with design by a senior engineer in terms of design time, chip performance, and gate count. An inference engine chip for real-time rule processing is used as a design example. KBSC is also compared with other logic synthesis systems.

ASIC automatic logic synthesis flowchart input form rule based expert systems silicon compilation

1 INTRODUCTION

Cost per large-scale integrated (LSI) chip can be defined by development cost D, fabrication cost F, and total production volume N. Therefore, chip cost C can be estimated by $C = D/N + F(N)$, where F is the function of N. If LSI circuits are produced in large volume, the development cost per chip is negligible. LSI circuits of this type include memories and other standard products.

ASICs are produced in small volume to meet customers' needs. In the case of ASICs, however, it becomes important to reduce development cost or design time. The process of ASIC design is divided into functional, logic, and layout stages. To reduce time required for functional and logic design, it is important to

1. Provide an optimum input form to specify an initial chip function.
2. Verify an initial chip function to ensure functional correctness.
3. Automatically translate a verified chip function to a logic circuit or netlist.

Correspondence and requests for reprints should be sent to Hideaki Kobayashi, International Chip Corporation, AT & T Building, 1201 Main St., Suite 2000, Columbia, SC 29201.

■ Manuscript received November 15, 1988; Manuscript revised June 5, 1989.

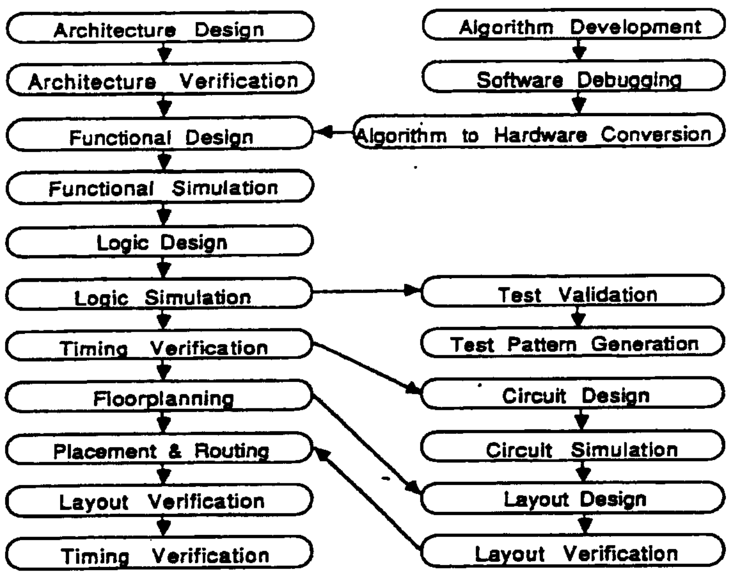


Figure 1. ASIC Development Flow.

Furthermore, to reduce time required for layout design, automatic translation of logic circuit (netlist) information to geometrical mask data is needed.

Many non-commercial logic synthesis systems [1–8] have been developed to translate automatically a behavioral or functional description into a logic circuit or chip layout. The concept of a knowledge-based silicon compiler (KBSC) was introduced in 1987 [9]. KBSC was developed jointly by Ricoh Company and International Chip Corporation (ICC). IF–THEN-type rules have been extracted from expert ASIC designers over several years at Ricoh and ICC, and stored in a knowledge base within KBSC. KBSC and other CAD tools are integrated into a comprehensive CAD system that provides designers with optimum input forms, such as flowcharts, state-transition equations, and functional module diagrams for cell-based design.

2 DESIGN METHODOLOGY

A typical ASIC development flow is shown in Figure 1. First, an entire system function is partitioned into a set of subfunctions (functional blocks). These functional blocks are then defined by various input forms, Figure 2.

1. Boolean expression or truth table form is for combinatorial circuit design.

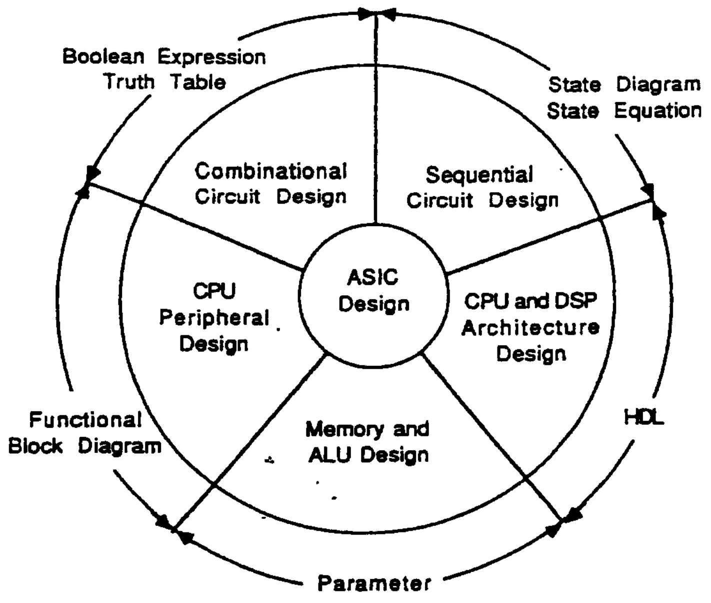


Figure 2. Input Forms for ASIC Design.

2. State-transition diagram or state-equation input form for sequential circuit design.
3. Parameter input form for memory and arithmetic and logic unit (ALU) design with variable address and data lengths.
4. Functional module input form using preregistered cells with bus compatibility for CPU peripheral LSI design.
5. Hardware description language (HDL) input form for CPU and digital signal processing (DSP) architecture design and performance evaluation.

Inputs 1 and 2 above require synthesis and optimization of logic. Logic synthesis and optimization are not required for input 3 since modules are generated by design rules and other constraints. Input 4 utilizes existing logic circuits corresponding to every module. Logic synthesis and optimization are not required since these circuits are already optimized. It is important to select an appropriate input form to design each functional block. The entire system design is completed by interconnecting these functional blocks.

KBSC's flowchart input form (Figure 4) is vital for "system" design that includes both data-path and control logic synthesis. KBSC flowcharts are useful for designers who are familiar with software programming as well as hardware system design. Flowchart-to-netlist conversion is achieved by an expert system where IF–THEN-type rules for logic synthesis are stored in a knowledge base. These rules are extracted from expert ASIC designers at Ricoh and ICC. KBSC's cell-based design approach provides quick turnaround time and design flexibility. Logic circuits designed by KBSC contain previously registered cells, each with information on its own mask data. Therefore, only placement and routing of these cells need to be performed by layout tools.

Layout design is performed in a hierarchical manner. First, highest level cells (functional blocks) are placed on a chip. An optimum placement is achieved by our floor planner, based on information such as each block area (transistor count), *X/Y* dimension ratio, and a netlist between blocks. The floor planner computes relative *X/Y* coordinates, routing areas between blocks, and information about terminal locations in each block.

After initial placement (floor planning), automatic routing takes place. Our automatic place–route software can handle macrocells with four-sided terminals as well as cells with only upper and lower terminals. Savings of 20% or more of the chip area are achieved by this approach compared with our conventional place–route software for two-sided terminals. To achieve 100% routability, routing area is estimated by heuristics. Extra routing area is eliminated by compaction. Layout of lower-level blocks is performed in a similar manner.

A data base stores cells with circuit information and physical mask data. Each cell contains information on logic symbols for schematic capture, models for logic stimulation, terminal names and locations for automatic placement and routing, cell sizes, and physical mask data. These cells are used for automatic logic synthesis as well as schematic capture to design logic circuits.

3 SYSTEM CONFIGURATION

The system configuration of KBSC is shown in Figure 3. The function of each software module in KBSC is explained.

The Flowchart Editor is an interactive functional editor for creation and modification of KBSC flowcharts for design specification entry. An example of a KBSC flowchart is shown in Figure 4. Actions and conditions as well as state transitions are represented by functional macros, which are independent of hardware. These macros are used to define data and numerical operators (e.g., add, subtract, shift, logical AND, logical OR) in each state. A sample list of macros is given in Figure 5.

The Flowchart Simulator is a verification tool for edited flowcharts at the functional level. It guarantees that edited flowcharts represent correct functions. ASIC users easily can operate the Flowchart Simulator to verify functional correctness.

After simulation, flowchart information is separated into actions and conditions. Actions are used for data-path synthesis, and conditions are used for control logic synthesis. Both actions and conditions are described

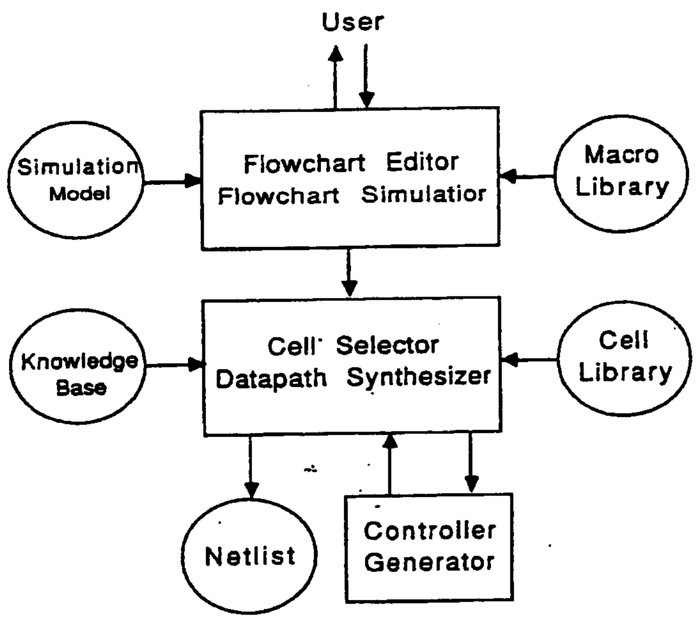
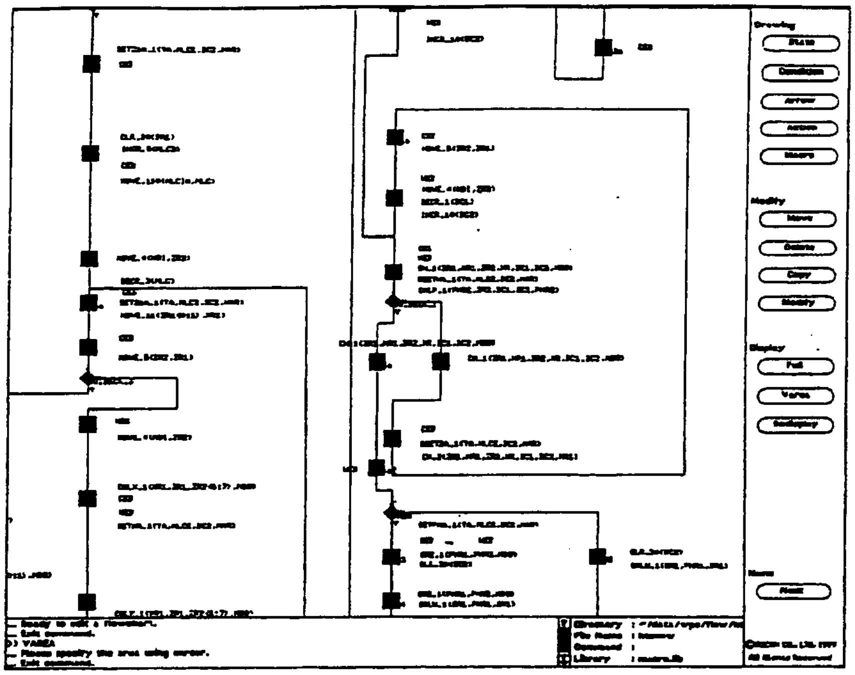


Figure 3. KBSC Configuration.

using the antecedent action form (AAF). An example of AAF is shown in Figure 6.

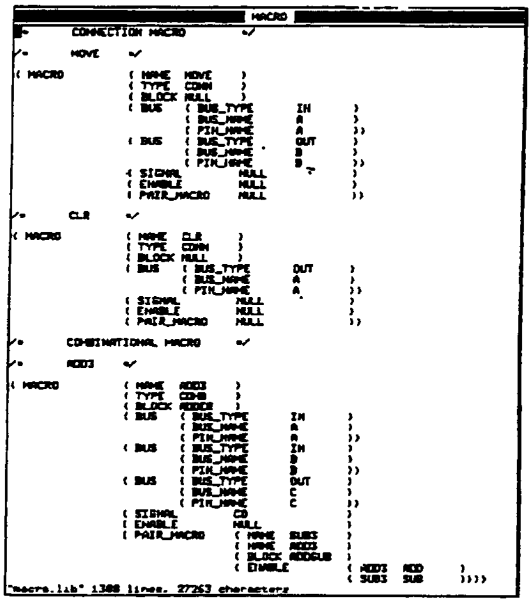
The Datapath Synthesizer provides automatic logic synthesis and optimization for data-path circuits. Synthesis and optimization of logic are accomplished by applying IF–THEN-type rules. Rules for placement and routing must be prepared for all macros. Example procedures for synthesis and optimization follow:

Place all macros according to rules for placement.
Place register blocks for all buses.
Route between all macros and register blocks. Add control signals to state-transition equations if needed.
Convert appropriate registers to counters and shifters. Add control signals to state-transition equations if needed.
Eliminate unnecessary macros and register blocks by classifying all macros and register blocks in terms of function and timing. Insert multiplexers into commonly used macros and register blocks, and add control signals to state-transition equations.
Connect clock signals. Clock signals for data-path circuits are synthesized separately from clock signals for control circuits. This is to assure that enable signals are stabilized before starting data transfer from data-path circuits.



```
Ready to edit a flowchart.
Exit command.
> VAREA
Please specify the area using cursor.
Exit command.
```

Figure 4. Example of a KBSC Flowchart.



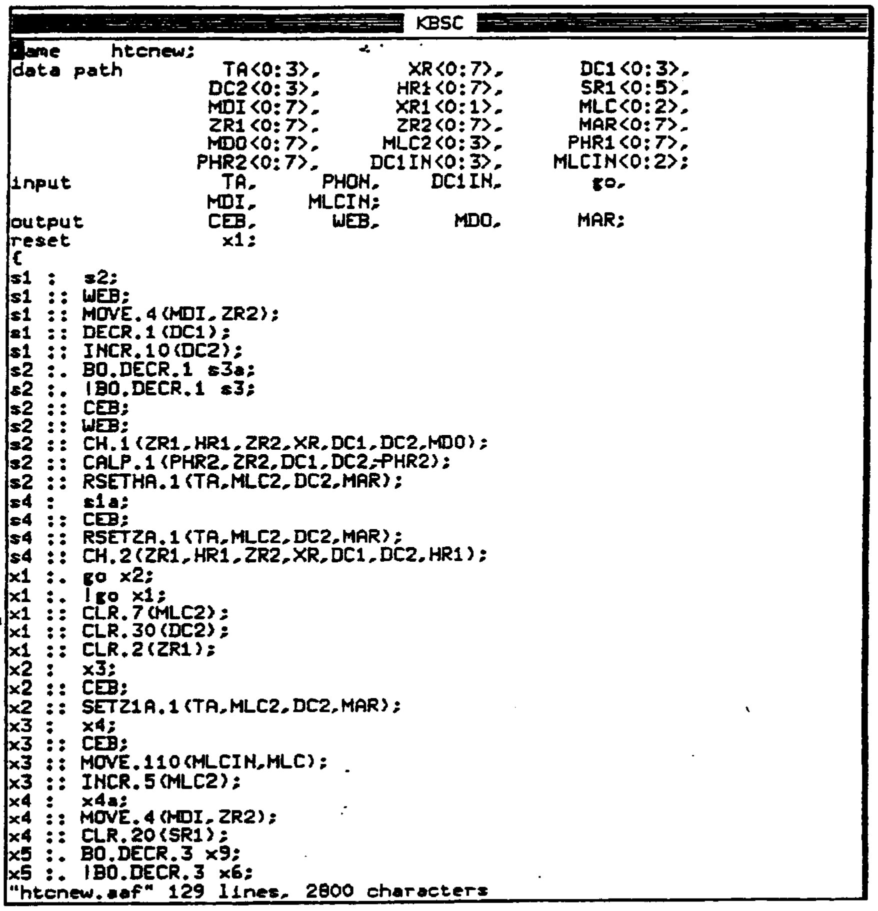
```
                              MACRO
/*      CONNECTION MACRO          */

/*      MOVE       */

( MACRO           ( NAME   MOVE      )
                  ( TYPE   CONN      )
                  ( BLOCK  NULL      )
                  ( BUS    ( BUS_TYPE          IN        )
                           ( BUS_NAME          A         )
                           ( PIN_NAME          A         ))
                  ( BUS    ( BUS_TYPE          OUT       )
                           ( BUS_NAME          B         )
                           ( PIN_NAME          B         ))
                  ( SIGNAL          NULL                 )
                  ( ENABLE          NULL                 )
                  ( PAIR_MACRO      NULL                 ))

/*      CLR        */

( MACRO           ( NAME   CLR       )
                  ( TYPE   CONN      )
                  ( BLOCK  NULL      )
                  ( BUS    ( BUS_TYPE          OUT       )
                           ( BUS_NAME          A         )
                           ( PIN_NAME          A         ))
                  ( SIGNAL          NULL                 )
                  ( ENABLE          NULL                 )
                  ( PAIR_MACRO      NULL                 ))

/*      COMBINATIONAL MACRO       */

/*      ADD3       */

( MACRO           ( NAME   ADD3      )
                  ( TYPE   COMB      )
                  ( BLOCK  ADDER     )
                  ( BUS    ( BUS_TYPE          IN        )
                           ( BUS_NAME          A         )
                           ( PIN_NAME          A         ))
                  ( BUS    ( BUS_TYPE          IN        )
                           ( BUS_NAME          B         )
                           ( PIN_NAME          B         ))
                  ( BUS    ( BUS_TYPE          OUT       )
                           ( BUS_NAME          C         )
                           ( PIN_NAME          C         ))
                  ( SIGNAL          CO                   )
                  ( ENABLE          NULL                 )
                  ( PAIR_MACRO      ( NAME   SUB3        )
                                    ( NAME   ADD3        )
                                    ( BLOCK  ADDSUB      )
                                    ( ENABLE             ( ADD3   ADD      )
                                                         ( SUB3   SUB      ))))
"macro.lib" 1308 lines, 27263 characters
```

Figure 5. Sample List of Macros.

The Cell Selector uses rules to select existing cells from a library to replace functional cells (without geometrical information) used in data-path synthesis. Example procedures for cell selection follow:

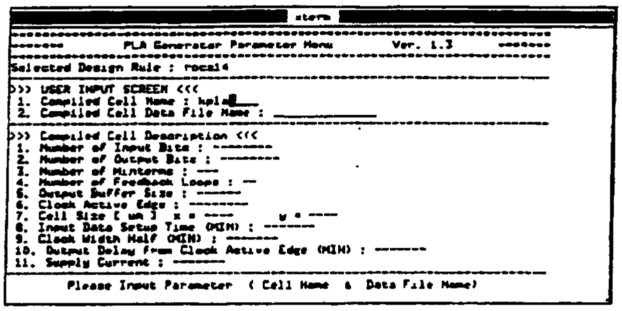
Enter design constraints such as speed, power consumption, and chip area.
Select cells that satisfy entered design constraints.
Eliminate unnecessary circuit portions if a selected cell has unused terminal(s).

The Controller Generator accepts state-transition equations (including modified state-transition equations during data-path synthesis) and auto-



```
name     htcnew;
data path            TA<0:3>,        XR<0:7>,        DC1<0:3>,
                    DC2<0:3>,       HR1<0:7>,        SR1<0:5>,
                    MDI<0:7>,       XR1<0:1>,        MLC<0:2>,
                    ZR1<0:7>,       ZR2<0:7>,        MAR<0:7>,
                    MDO<0:7>,      MLC2<0:3>,       PHR1<0:7>,
                   PHR2<0:7>,    DC1IN<0:3>,      MLCIN<0:2>;
input                TA,      PHON,     DC1IN,       go,
                    MDI,     MLCIN;
output              CEB,       WEB,       MDO,       MAR;
reset                x1;
{
s1 :  s2;
s1 :: WEB;
s1 :: MOVE.4(MDI,ZR2);
s1 :: DECR.1(DC1);
s1 :: INCR.10(DC2);
s2 :. BO.DECR.1 s3a;
s2 :. !BO.DECR.1 s3;
s2 :: CEB;
s2 :: WEB;
s2 :: CH.1(ZR1,HR1,ZR2,XR,DC1,DC2,MDO);
s2 :: CALP.1(PHR2,ZR2,DC1,DC2,PHR2);
s2 :: RSETHA.1(TA,MLC2,DC2,MAR);
s4 :  s1a;
s4 :: CEB;
s4 :: RSETZA.1(TA,MLC2,DC2,MAR);
s4 :: CH.2(ZR1,HR1,ZR2,XR,DC1,DC2,HR1);
x1 :. go x2;
x1 :. !go x1;
x1 :: CLR.7(MLC2);
x1 :: CLR.30(DC2);
x1 :: CLR.2(ZR1);
x2 :  x3;
x2 :: CEB;
x2 :: SETZ1A.1(TA,MLC2,DC2,MAR);
x3 :  x4;
x3 :: CEB;
x3 :: MOVE.110(MLCIN,MLC);
x3 :: INCR.5(MLC2);
x4 :  x4a;
x4 :: MOVE.4(MDI,ZR2);
x4 :: CLR.20(SR1);
x5 :. BO.DECR.3 x9;
x5 :. !BO.DECR.3 x6;
"htcnew.aaf" 129 lines, 2800 characters
```

Figure 6. Example of Antecendent Action Form.



```
                                  xterm
========================================================================
=======         PLA Generator Parameter Menu          Ver. 1.3   =======
========================================================================
Selected Design Rule : rocal4
------------------------------------------------------------------------
>>> USER INPUT SCREEN <<<
 1. Compiled Cell Name : kpla____
 2. Compiled Cell Data File Name : ______________
------------------------------------------------------------------------
>>> Compiled Cell Description <<<
 1. Number of Input Bits : --------
 2. Number of Output Bits : --------
 3. Number of Minterms : ---
 4. Number of Feedback Loops : --
 5. Output Buffer Size : ------
 6. Clock Active Edge : --------
 7. Cell Size [ um ]  x = ----       y = ----
 8. Input Data Setup Time (MIN) : -------
 9. Clock Width Half (MIN) : -------
 10. Output Delay from Clock Active Edge (MIN) : -------
 11. Supply Current : -------
------------------------------------------------------------------------
        Please Input Parameter  ( Cell Name  &  Data File Name)
```

Figure 7. Parameter Menu for PLA Generation.

matically performs synthesis and optimization of logic circuits. Synthesis and optimization can be done in the following two stages.

1. Make a state assignment from state-transition equations and obtain Boolean expressions for circuits that implement state codes and output functions. Example procedures for making a state assignment follow:

   Count the total number of states and calculate the number of bits required to represent each state code.
   Assign a unique code for each state in order of appearance.
   Obtain Boolean expressions from a truth table describing present states, conditions, and next states.

2. Minimize Boolean expressions and generate a parameter menu for PLA generation (Figure 7), or generate a netlist for a logic circuit composed of existing cells. Example procedures for synthesizing a PLA-based system controller follow:

   Derive a truth table from Boolean expressions.
   Reduce the truth table by applying rules for data compression. Obtain a parameter file for PLA generation.

   Example procedures for system controller synthesis using random logic follow:

   Convert Boolean expressions to logic using only NAND gates.
   Minimize logic by applying rules for optimization.
   Convert logic to a circuit using NAND and/or NOR gates with less than or equal to six inputs each.
   Minimize logic by applying rules for optimization.
   Perform an error check for fan-out number excess and other checks.

   A logic circuit (netlist) is generated automatically by combining optimized data-path and control circuits. An example of a logic circuit generated by KBSC is shown in Figure 8.

4 RULES FOR DATA-PATH SYNTHESIS

Example rule numbers used to synthesize data-paths are shown in Table 1 in execution (firing) order. The number of execution times for each rule is also shown in Table 2. Rule descriptions for data-path synthesis follow:

Rule 10 connects clock and reset terminals to data path circuits.
Rule 100 places an input buffer for an external input bus with a corresponding data length. It also places a register for an internal bus.
Rule 200 places macros (add, sub, and, or). If the macro's first operand is a bus, then it places a block with a corresponding data length. For example, add 0.12 (OP < 16:31 > C1) places a block with 16 bits of data length.
Rule 202 places macros (not, sfr, sfl). Similar to Rule 200.
Rule 220 places a macro (cmp). Similar to Rule 200.
Rule 230 places output buffers.

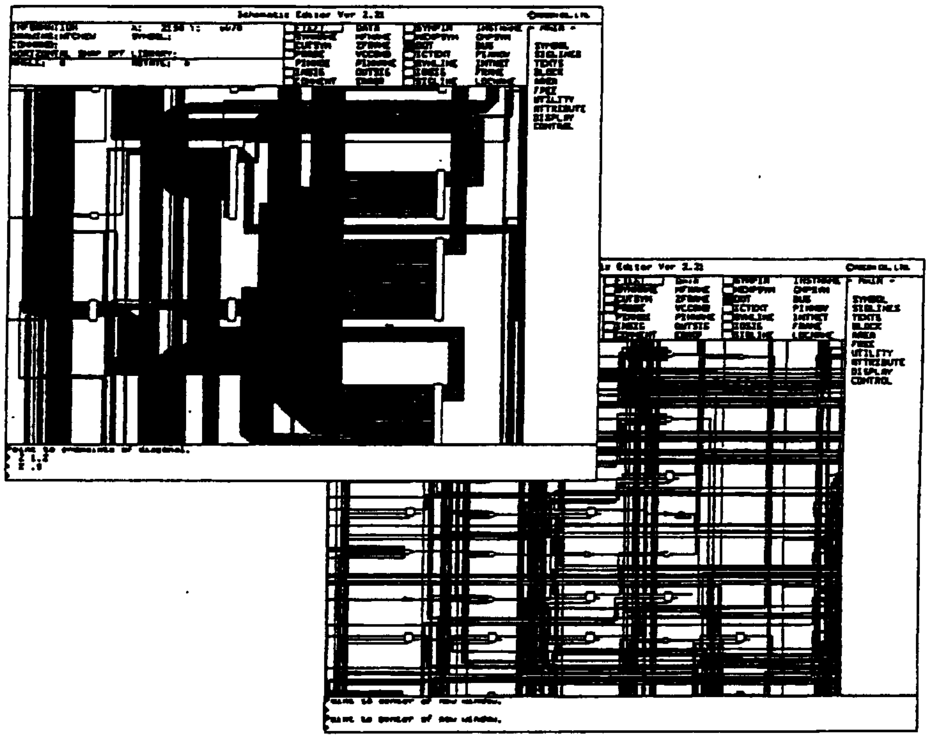


Figure 8. Example of a Logic Circuit Synthesized by KBSC.

Table 1. Rule Firing Sequence for Data-path Synthesis

CONTEXT	BEGIN	RULE 10
CONTEXT	BUSBLOCK	RULE 100
CONTEXT	MACROBLOCK	RULE 200
CONTEXT	MACROBLOCK	RULE 202
CONTEXT	MACROBLOCK	RULE 210
CONTEXT	MACROBLOCK	RULE 220
CONTEXT	MACROBLOCK	RULE 230
CONTEXT	MACROBLOCK	RULE 240
CONTEXT	CONNECTION	RULE 302
CONTEXT	CONNECTION	RULE 308
CONTEXT	CONNECTION	RULE 320
CONTEXT	CONNECTION	RULE 330
CONTEXT	CONNECTION	RULE 332
CONTEXT	CONNECTION	RULE 340
CONTEXT	CONNECTION	RULE 334
CONTEXT	CONNECTION	RULE 310
CONTEXT	CONNECTION	RULE 350
CONTEXT	CONNECTION	RULE 360
CONTEXT	CONNECTION	RULE 370
CONTEXT	CONNECTION	RULE 372
CONTEXT	CONNECTION	RULE 380
CONTEXT	TRANSFORM	RULE 400
CONTEXT	TRANSFORM	RULE 410
CONTEXT	ADDCLOCK	RULE 500
CONTEXT	ADDCLOCK	RULE 510
CONTEXT	ADDCLOCK	RULE 520
CONTEXT	SELECTION	RULE 610
CONTEXT	SELECTION	RULE 620
CONTEXT	SELECTION	RULE 630
CONTEXT	SELECTION	RULE 640
CONTEXT	SELECTION	RULE 650
CONTEXT	SELECTION	RULE 655
CONTEXT	SELECTION	RULE 660
CONTEXT	SELECTION	RULE 665
CONTEXT	SELECTION	RULE 670
CONTEXT	SELECTION	RULE 675
CONTEXT	SELECTION	RULE 680
CONTEXT	SELECTION	RULE 685
CONTEXT	SELECTION	RULE 642
CONTEXT	SELECTION	RULE 644
CONTEXT	SELECTION	RULE 600

Table 2. Number of Executions for Each Rule

RULE	10	TIME	1
RULE	100	TIME	1
RULE	200	TIME	2
RULE	202	TIME	3
RULE	210	TIME	1
RULE	220	TIME	1
RULE	230	TIME	4
RULE	240	TIME	1
RULE	300	TIME	1
RULE	302	TIME	1
RULE	308	TIME	3
RULE	310	TIME	1
RULE	320	TIME	1
RULE	330	TIME	9
RULE	332	TIME	4
RULE	334	TIME	1
RULE	340	TIME	1
RULE	350	TIME	4
RULE	360	TIME	1
RULE	370	TIME	3
RULE	372	TIME	7
RULE	380	TIME	10
RULE	400	TIME	2
RULE	410	TIME	1
RULE	500	TIME	2
RULE	510	TIME	1
RULE	520	TIME	8
RULE	600	TIME	1
RULE	610	TIME	8
RULE	620	TIME	5
RULE	630	TIME	2
RULE	640	TIME	1
RULE	642	TIME	3
RULE	644	TIME	4
RULE	650	TIME	1
RULE	655	TIME	1
RULE	660	TIME	3
RULE	665	TIME	1
RULE	670	TIME	3
RULE	675	TIME	1
RULE	680	TIME	3
RULE	685	TIME	1

Case 5:03-cv-04669-JW Document 243-3 Filed 09/10/2004 Page 20 of 23

Rule 240 places input buffers.
Rule 210 clears all flags to change mode from placement to routing.
Rule 300 routes macros (add, sub, and, or). It connects a bus register's output terminal, corresponding to the macro's first operand, to the macro's "A" terminal. Similarly, it applies to the macro's second and third operands.
Rule 302 is similar to Rule 300. It applies to the macro's second operand.

A total of 114 rules (42 different types of rules) are applied to synthesize data-paths automatically and to convert them to an equivalent circuit composed of existing cells.

5 RULES FOR CONTROL LOGIC SYNTHESIS

Example rule numbers used to synthesize control logic are shown in Table 3 in execution (firing) order. Rule descriptions for control logic synthesis follow:

Rule 10 constructs a truth table from Boolean expressions.
Rule 100 merges two rows in a truth table with the same combination of inputs into a single row.
Rule 500 checks the fan-out.

Table 3. Rule Firing Sequence for Control Logic Synthesis

CONTEXT	BEGIN	RULE 10	CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 450
CONTEXT	BEGIN	RULE 20	CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 480
CONTEXT	MINIMIZE	RULE 100	CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 450
CONTEXT	MINIMIZE	RULE 130	CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 450
CONTEXT	MINIMIZE	RULE 100	CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 450
CONTEXT	MINIMIZE	RULE 130	CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 450
CONTEXT	MINIMIZE	RULE 130	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	OUT	RULE 210	CONTEXT	MDLMIN	RULE 400	CONTEXT	ERRCHK	RULE 500
CONTEXT	MINIMIZE	RULE 130	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	OUT	RULE 200	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 300	CONTEXT	MDLMIN	RULE 400	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 430	CONTEXT	MDLMIN	RULE 400	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 500
CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 460	CONTEXT	ERRCHK	RULE 510
CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 460	CONTEXT	ADDBUF	RULE 600
CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 460	CONTEXT	ADDBUF	RULE 600
CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 470	CONTEXT	ADDBUF	RULE 600
CONTEXT	MDLMIN	RULE 450	CONTEXT	MDLMIN	RULE 440	CONTEXT	ADDBUF	RULE 600
CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 450	CONTEXT	ADDBUF	RULE 600
CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 450	CONTEXT	ADDBUF	RULE 600
CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 450	CONTEXT	ADDBUF	RULE 600
CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 450	CONTEXT	ADDBUF	RULE 610
CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 480			
CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 450			
CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 450			
CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 450			
CONTEXT	MDLMIN	RULE 460	CONTEXT	MDLMIN	RULE 480			
CONTEXT	MDLMIN	RULE 400	CONTEXT	MDLMIN	RULE 450			
			CONTEXT	MDLMIN	RULE 450			

A total of 110 rules (18 different types of rules) are applied to synthesize control logic automatically and to convert it to a circuit using existing cells.

6 PERFORMANCE EVALUATION

ASIC design by KBSC was compared with design by a senior engineer in terms of design time, chip performance, and gate count. An inference engine chip for real-time rule processing is used as a design example.

1. *Design time:* KBSC needed only one-fifth of the design time required by a senior engineer. KBSC took approximately three weeks to complete the chip layout, whereas a senior engineer took approximately 15 weeks. Most of the time spent by the engineer was for architecture design.
2. *Chip performance:* The number of cycles to execute an IF-THEN-type rule is defined as a unit parameter. One rule was executed with 13 cycles for the circuit designed by KBSC, whereas the circuit designed by the senior engineer took 8 cycles to execute one rule. This was mainly due to the engineer's experience in designing circuits that can process more than two pieces of data in parallel.
3. *Gate count:* The KBSC circuit required 2,800 gates; the circuit designed by the engineer required 2,500 gates. The difference of 300 gates is due to the lack of rules for eliminating unused gates and optimizing logic circuits.

7 COMPARISON BETWEEN LOGIC SYNTHESIZERS

Logic synthesis systems in References [1–8] are compared with KBSC in terms of flowchart input form and rule-based approach to automatic data-path and control logic synthesis. A technology-oriented register transfer (RT)-level description is used in [1]. Input to KBSC is not an RT-level description, as used in many other approaches. KBSC's input is also not a flowchart like specification of control for typical finite-state machine design. Rules used in KBSC do synthesize both data-path and control circuits from a hardware-independent flowchart description. Output from KBSC is a netlist for automatic placement and routing.

Input to ALERT [2, 3] is an architectural description in Iverson's APL notation with declarations of variables to represent physical devices such as flip-flops and registers. The user of ALERT needs to specify memory size, word length, instruction format, and other hardware design features. In contrast, KBSC's input is a flowchart description with functional macros that are independent of hardware.

Input to the CMU-DA system [4] is an algorithmic description in ISP language. ISP description is translated to the first-level path graph with interconnections of abstract components. It is then mapped to the second level of the data-path graph with selected physical modules.

KBSC001129

The Design Automation Assistant (DAA) [5] is an expert system that uses heuristic rules to synthesize architectural implementation from an algorithmic description with design constraints. Input to DAA is written in ISPS, a description for a Digital Equipment Corporation computer. The DAA produces a hardware network composed of modules, ports, links, and symbolic microcode.

Input to Flamel [6] is a behavioral description in the form of a Pascal program. Flamel's input is associated with a specified bus architecture. Functional blocks such as ALUs, registers, and I/O pads are ordered and placed on buses. Multiplexers are needed to regulate bus traffic at each block's input and output. In contrast, the KBSC approach is not tied to any fixed bus architecture and needs no multiplexers to regulate bus traffic.

SOCRATES [7] uses an expert system to optimize combinatorial logic for a specific target technology. Rules used in SOCRATES optimize gate-level circuits for speed and area in a given technology. A rule replaces a portion of a circuit by a functionally equivalent but more optimal circuit.

8 CONCLUSION

It has been shown that a knowledge-based silicon compiler (KBSC) dramatically reduces time required for functional and logic design. KBSC is clearly distinguished from other logic synthesis systems in terms of its flowchart input form and rule-based approach to automatic data-path and control logic synthesis. More than 10 chips have been designed using KBSC. Applications of these chips include speech synthesis, rule processing, vector-raster conversion, Japanese-character optical character recognition, and printer control.

From these initial results, we believe that KBSC is a useful tool for designers who are not familiar with hardware or circuit design. To improve the current version of KBSC more rules are needed for parallel processing and logic optimization. Most of the problems observed can be solved by simply adding new rules to the knowledge base and modifying rules. The program is written in C language and runs on Sun and other workstations supporting the UNIX operating system and the X windows system.

REFERENCES

[1] J.A. Darringer, and W.H. Joyner, Jr., "A New Look at Logic Synthesis," *Proc. of 17th Design Automation Conf.*, pp. 543–549, 1980.

[2] T.D. Friedman, and S.C. Yang, "Methods Used in an Automatic Logic Design Generator (ALERT)," *IEEE Transactions on Computers*, Vol. C-18, No. 7, pp. 593–614, July 1969.

[3] T.D. Friedman, and S.C. Yang, "Quality of Designs from an Automatic Logic Generator (ALERT)," *Proc. of 7th Design Automation Conf.*, pp. 71–89, 1970.

[4] A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, and J. Kim, "The CMU Design Automation System—An Example of Automated Data Path

Design," *Proc. of 16th Design Automation Conf.*, Las Vegas, NV, pp. 73–80, 1979.

[5] T.J. Kowalski, D.J. Geiger, W.H. Wolf, and W. Fichter, "The VLSI Design Automation Assistant: From Algorithms to Silicon," *IEEE Design and Test of Computers Magazine*, Vol. 2, No. 4, pp. 33–43, August 1986.

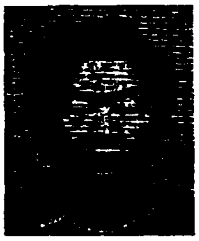
[6] H. Trickey, "Flamel: A High-Level Hardware Compiler," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-6, No. 2, pp. 259–269, March 1987.

[7] A.J. deGeus, and W. Cohen, "A Rule-Based System for Optimizing Combinational Logic," *IEEE Design and Test of Computers Magazine*, Vol. 2, No. 4, pp. 22–32, August 1986.

[8] L. Trevillyan, "An Overview of Logic Synthesis Systems," *Proc. of 24th ACM/IEEE Design Automation Conf.*, Paper 9.1, pp. 166–172, 1987.

[9] H. Kobayashi, "KBSC: A Knowledge Based Silicon Compiler—A Future Trend in ASIC Design," Ricoh ASIC Technical Seminar, Tokyo, Japan, October, 1987.

Hideaki Kobayashi (See guest editorial section (p. 355) of this special issue for author's biography).



Terumi Suehiro joined Ricoh Company in 1979 and has been involved in the development of CPU peripheral LSI. Since 1985 he has been developing a Ricoh CAD system for ASICs. He is currently Manager and Senior Engineer of the Semiconductors R & D Center. He received a B.S. degree in applied physics from Nagoya University in 1979.

Masahiro Shindo joined Ricoh Company in 1979 and has been involved in ASIC design, CAD tools, and strategic management. He is currently the Assistant General Manager of the Electronic Devices Division and Director of the Semiconductors R & D Center. Prior to joining Ricoh Company, he was employed by Mitsubishi Electric Corp. and was involved in the development of integrated circuit process technology (micron process, CVD) and devices (DRAM, SRAM, EPROM, LOGIC). He received a B.S. degree in industrial chemical engineering from the University of Ehime in 1963.

US005197016A

United States Patent [19]

Sugimoto et al.

[11] Patent Number: 5,197,016

[45] Date of Patent: Mar. 23, 1993

[54] INTEGRATED SILICON-SOFTWARE COMPILER

[75] Inventors: Tai Sugimoto; Hideaki Kobayashi, both of Columbia, S.C.; Masahiro Shindo; Haruo Nakayama, both of Osaka, Japan

[73] Assignees: International Chip Corporation, Columbia, S.C.; Ricoh Company, Ltd., Tokyo, Japan

[21] Appl. No.: 380,079

[22] Filed: Jul. 14, 1989

Related U.S. Application Data

[63] Continuation-in-part of Ser. No. 143,821, Jan. 13, 1988, Pat. No. 4,922,432.

[51] Int. Cl.^{5} G06F 15/60

[52] U.S. Cl. 364/490; 364/489; 364/488

[58] Field of Search 364/488, 489, 490, 491

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,648,044	3/1987	Hardy et al.	395/76
4,658,370	4/1987	Erman et al.	395/76
4,675,829	6/1987	Clemenson	395/65
4,703,435	10/1987	Darringer et al.	364/489
4,922,432	5/1990	Kobayashi et al.	364/490

OTHER PUBLICATIONS

"A Front End Graphic Interface to the First Silicon Compiler" by J. H. Nash et al., European Conf. on Electronic Design Automation (EDA84), pp. 120–124.

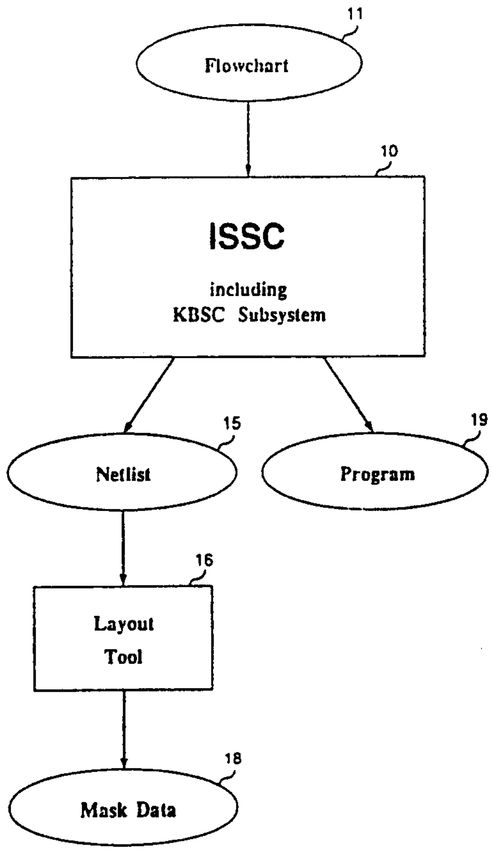
An Engineering Approach to Digital Design, William I. Fletcher, Prentice-Hall, Inc. pp. 491-505.

Primary Examiner—Vincent N. Trans
Attorney, Agent, or Firm—Bell, Seltzer, Park & Gibson

[57] **ABSTRACT**

A computer-aided system and method is disclosed for designing an application specific integrated circuit (ASIC) whose intended function is implemented both by a hardware subsystem including hardware elements on the integrated circuit and by a software subsystem including a general purpose microprocessor also on the integrated circuit. The system also generates software instructions for use by the software subsystem. The system utilizes a knowledge based expert system, with a knowledge base extracted from expert ASIC designers, and thus makes it possible for ASIC's to be designed and provided quickly and economically by persons not having the highly specialized skill of an ASIC designer.

32 Claims, 7 Drawing Sheets



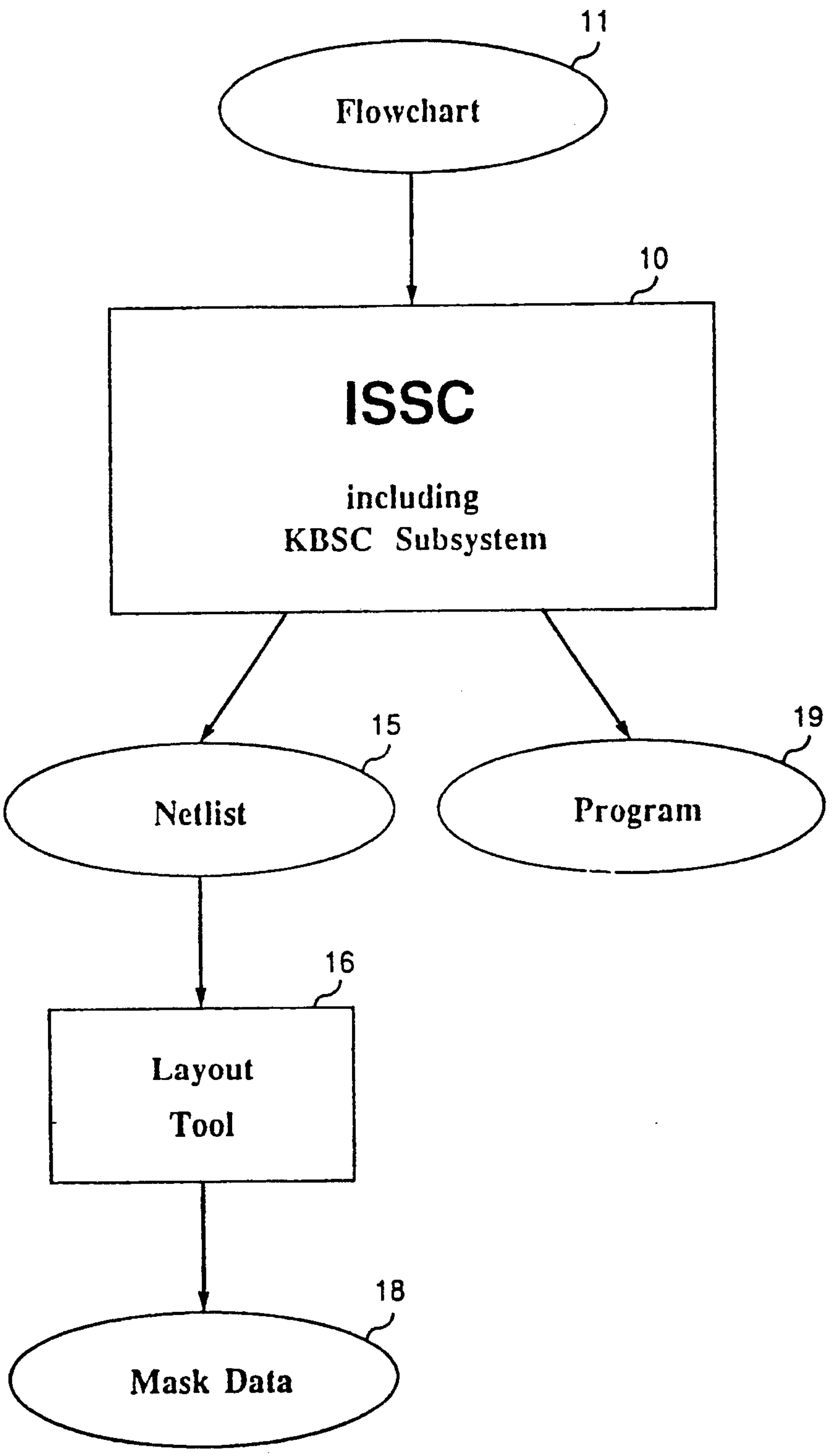

11
Flowchart
10
ISSC
including
KBSC Subsystem
15
Netlist
19
Program
16
Layout
Tool
18
Mask Data

FIG. 1.

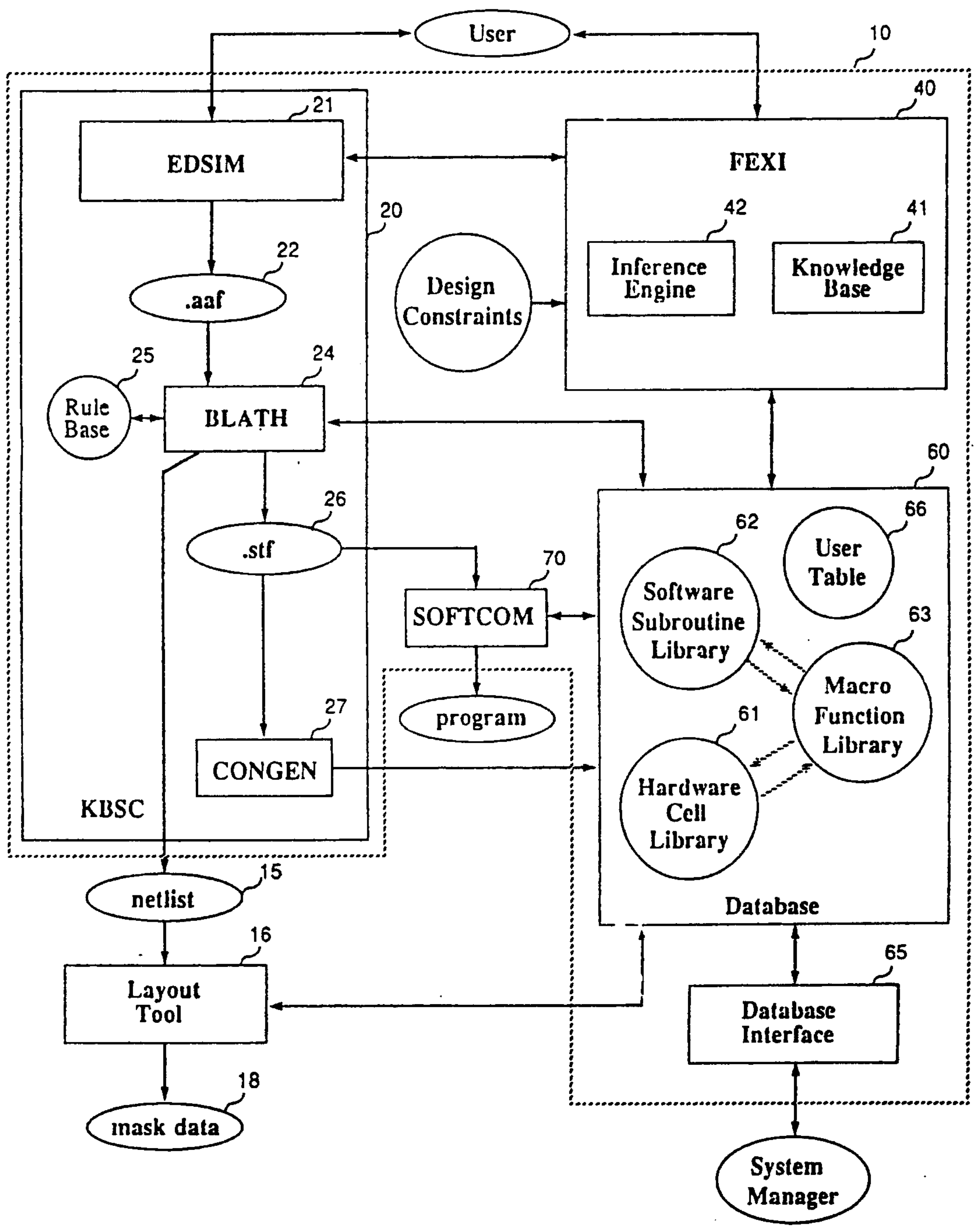

User
10
40
21
EDSIM
FEXI
20
42
41
22
.aaf
Design Constraints
Inference Engine
Knowledge Base
25
24
Rule Base
BLATH
60
26
.stf
62
66
70
User Table
SOFTCOM
Software Subroutine Library
63
program
Macro Function Library
27
61
CONGEN
Hardware Cell Library
KBSC
15
netlist
Database
16
65
Layout Tool
Database Interface
18
mask data
System Manager

FIG. 2.

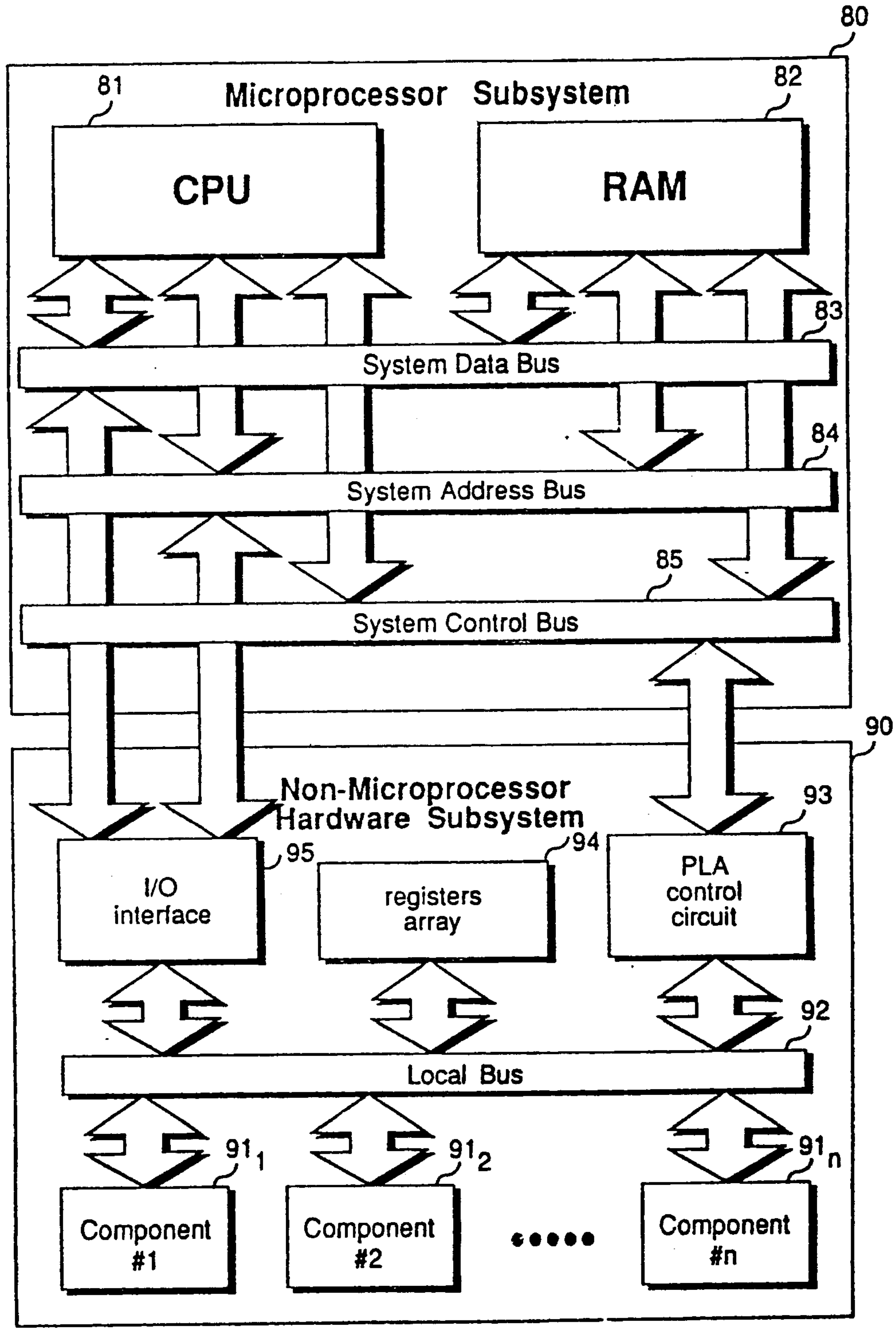


FIG. 3.

DEF017268

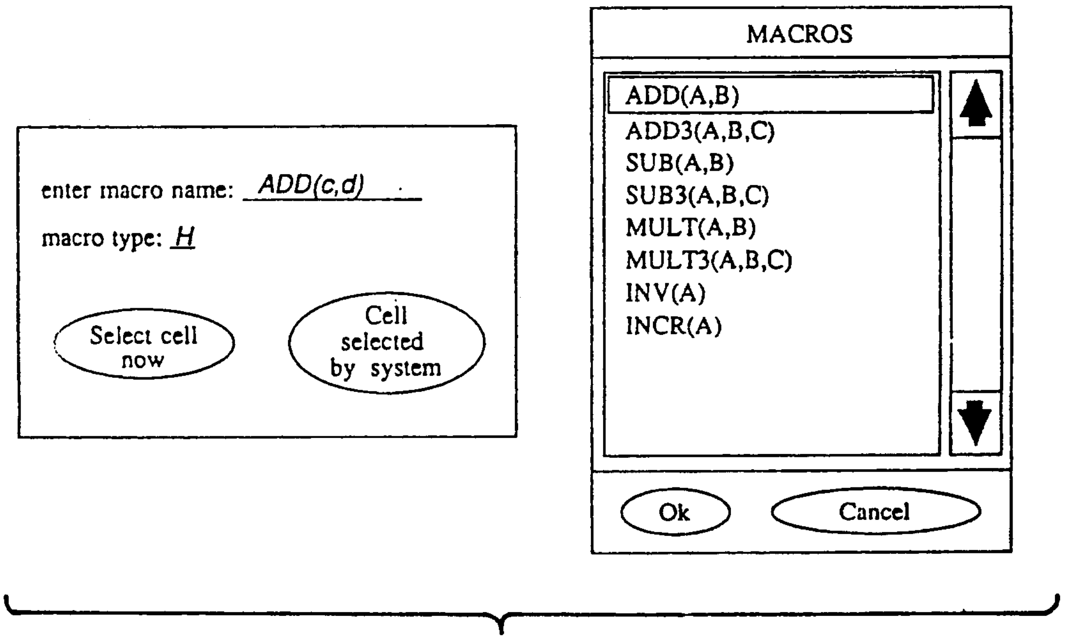

enter macro name: ADD(c,d)
macro type: H
Select cell now
Cell selected by system
MACROS
ADD(A,B)
ADD3(A,B,C)
SUB(A,B)
SUB3(A,B,C)
MULT(A,B)
MULT3(A,B,C)
INV(A)
INCR(A)
Ok
Cancel

FIG. 4.

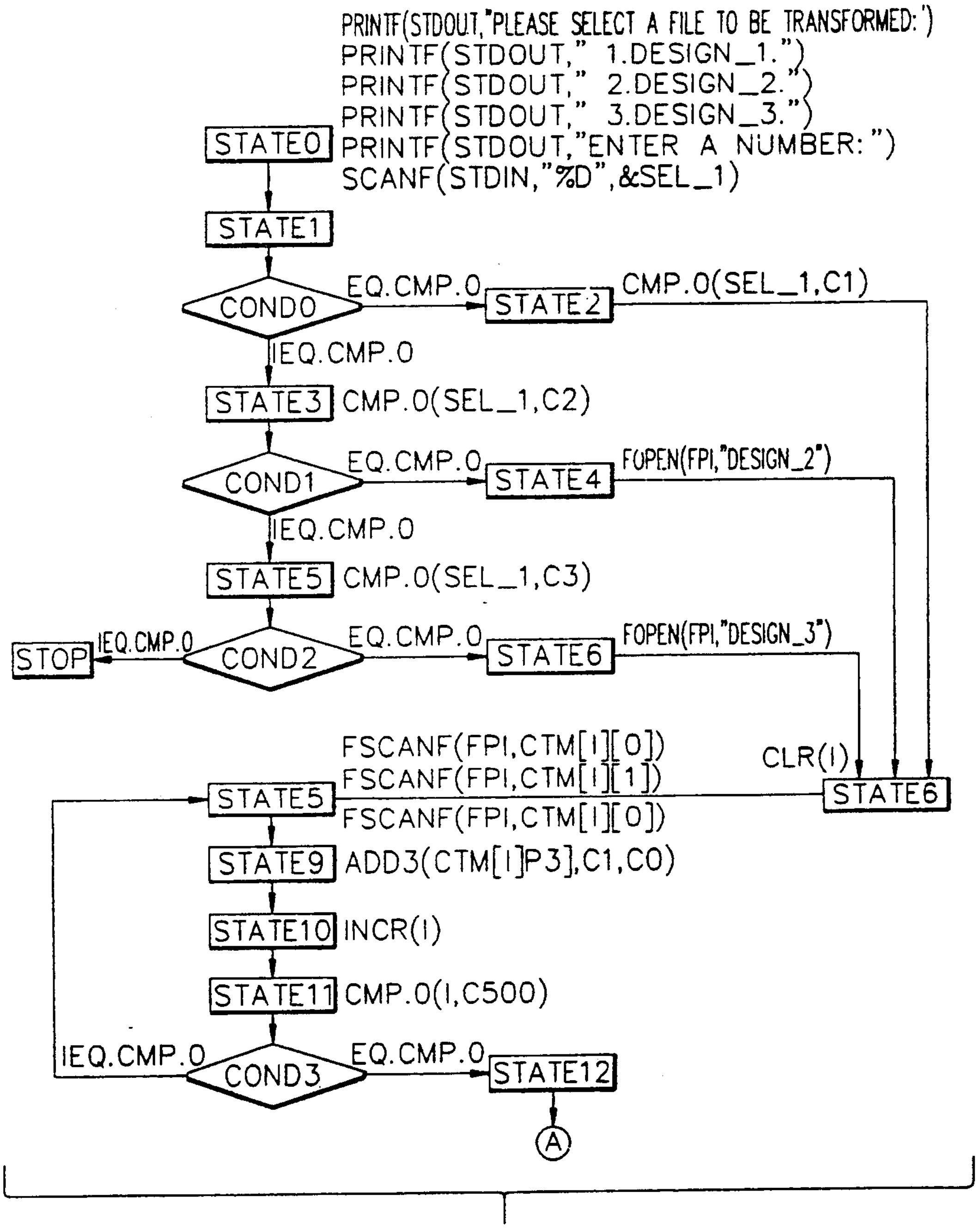

PRINTF(STDOUT,"PLEASE SELECT A FILE TO BE TRANSFORMED:')
PRINTF(STDOUT," 1.DESIGN_1.")
PRINTF(STDOUT," 2.DESIGN_2.")
PRINTF(STDOUT," 3.DESIGN_3.")
PRINTF(STDOUT,"ENTER A NUMBER:")
SCANF(STDIN,"%D",&SEL_1)
STATE0
STATE1
COND0
EQ.CMP.0
STATE2
CMP.0(SEL_1,C1)
IEQ.CMP.0
STATE3
CMP.0(SEL_1,C2)
COND1
EQ.CMP.0
STATE4
FOPEN(FPI,"DESIGN_2")
IEQ.CMP.0
STATE5
CMP.0(SEL_1,C3)
STOP
IEQ.CMP.0
COND2
EQ.CMP.0
STATE6
FOPEN(FPI,"DESIGN_3")
FSCANF(FPI,CTM[I][0])
FSCANF(FPI,CTM[I][1])
CLR(I)
STATE5
STATE6
FSCANF(FPI,CTM[I][0])
STATE9
ADD3(CTM[I]P3],C1,C0)
STATE10
INCR(I)
STATE11
CMP.0(I,C500)
IEQ.CMP.0
COND3
EQ.CMP.0
STATE12
A
TO FIG. 6.

FIG. 5.

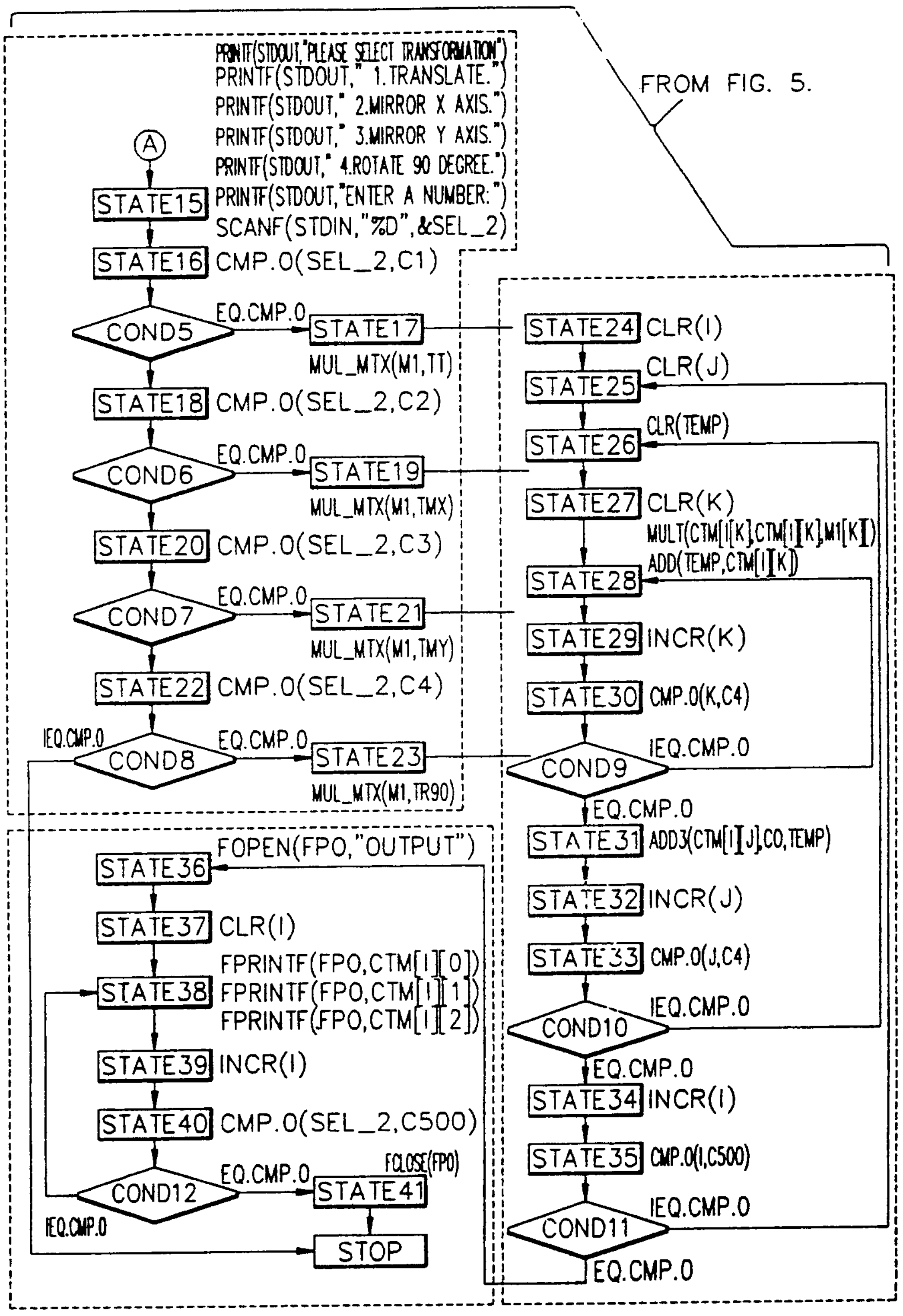

FROM FIG. 5.
A
STATE15
PRINTF(STDOUT,"PLEASE SELECT TRANSFORMATION")
PRINTF(STDOUT," 1.TRANSLATE.")
PRINTF(STDOUT," 2.MIRROR X AXIS.")
PRINTF(STDOUT," 3.MIRROR Y AXIS.")
PRINTF(STDOUT," 4.ROTATE 90 DEGREE.")
PRINTF(STDOUT,"ENTER A NUMBER:")
SCANF(STDIN,"%D",&SEL_2)
STATE16 CMP.0(SEL_2,C1)
COND5 EQ.CMP.0 STATE17 MUL_MTX(M1,TT)
STATE18 CMP.0(SEL_2,C2)
COND6 EQ.CMP.0 STATE19 MUL_MTX(M1,TMX)
STATE20 CMP.0(SEL_2,C3)
COND7 EQ.CMP.0 STATE21 MUL_MTX(M1,TMY)
STATE22 CMP.0(SEL_2,C4)
COND8 IEQ.CMP.0 EQ.CMP.0 STATE23 MUL_MTX(M1,TR90)
STATE24 CLR(I)
STATE25 CLR(J)
STATE26 CLR(TEMP)
STATE27 CLR(K)
MULT(CTM[I][K],CTM[I][K],M1[K][J])
ADD(TEMP,CTM[I][K])
STATE28
STATE29 INCR(K)
STATE30 CMP.0(K,C4)
COND9 IEQ.CMP.0 EQ.CMP.0
STATE31 ADD3(CTM[I][J],CO,TEMP)
STATE32 INCR(J)
STATE33 CMP.0(J,C4)
COND10 IEQ.CMP.0 EQ.CMP.0
STATE34 INCR(I)
STATE35 CMP.0(I,C500)
COND11 IEQ.CMP.0 EQ.CMP.0
STATE36 FOPEN(FPO,"OUTPUT")
STATE37 CLR(I)
STATE38 FPRINTF(FPO,CTM[I][0]) FPRINTF(FPO,CTM[I][1]) FPRINTF(FPO,CTM[I][2])
STATE39 INCR(I)
STATE40 CMP.0(SEL_2,C500)
COND12 IEQ.CMP.0 EQ.CMP.0
STATE41 FCLOSE(FPO)
STOP

FIG. 6.

DEF047271

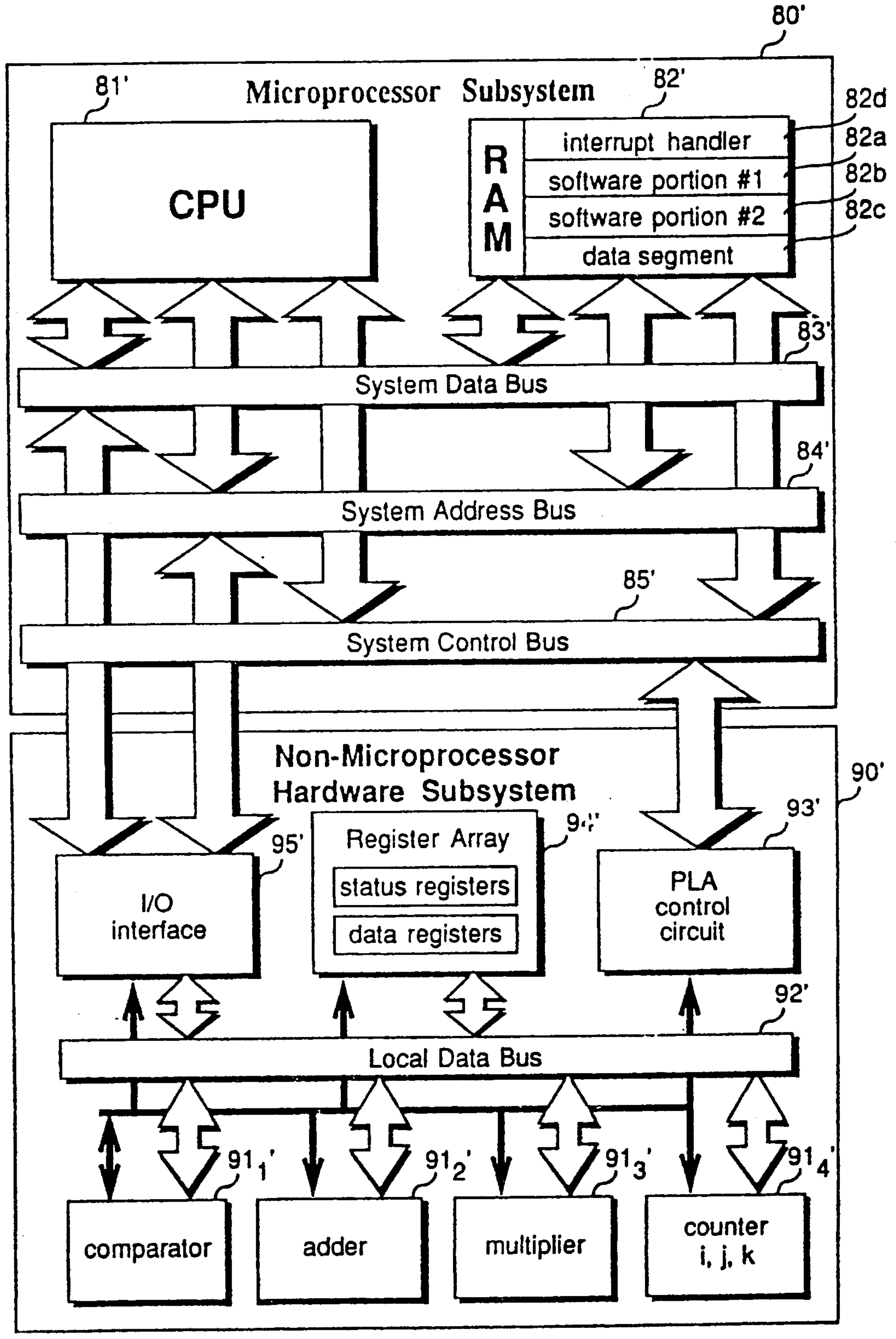


FIG. 7

DEF017272

INTEGRATED SILICON-SOFTWARE COMPILER

CROSS-REFERENCE TO RELATED APPLICATION

This application is a continuation-in-part of copending U.S. application Ser. No. 143,821, filed Jan. 13, 1988, now U.S. Pat. No. 4,922,432.

FIELD AND BACKGROUND OF THE INVENTION

This invention relates to the design of integrated circuits, and more particularly relates to a computer-aided system and method for designing application specific integrated circuits.

An application specific integrated circuit (ASIC) is an integrated circuit chip designed to perform a specific function, as distinguished from standard, general purpose integrated circuit chips, such as microprocessors, memory chips, etc. A highly skilled design engineer having specialized knowledge in VLSI circuit design is ordinarily required to design an ASIC. In the design process, the VLSI design engineer will consider the particular objectives to be accomplished and tasks to be performed by the integrated circuit and will create structural level design specifications which define the various hardware components required to perform the desired function, as well as the interconnection requirements between these components. A system controller must also be designed for synchronizing the operations of these components. This requires an extensive and all encompassing knowledge of the various hardware components required to achieve the desired objectives, as well as their interconnection requirements, signal level compatibility, timing compatibility, physical layout, etc. At each design step, the designer must do tedious analysis. The design specifications created by the VLSI design engineer may, for example, be in the form of circuit schematics, parameters or specialized hardware description languages (HDLs).

From the structural level design specifications, the description of the hardware components and interconnections is converted to a physical chip layout level description which describes the actual topological characteristics of the integrated circuit chip. This physical chip layout level description provides the mask data needed for fabricating the chip.

Due to the tremendous advances in very large scale integration (VLSI) technology, highly complex circuit systems are being built on a single chip. With their complexity and the demand to design custom chips at a faster rate, in large quantities, and for an ever increasing number of specific applications, computer-aided design (CAD) techniques need to be used. CAD techniques have been used with success in design and verification of integrated circuits, at both the structural level and at the physical layout level. For example, CAD systems have been developed for assisting in converting VLSI structural level descriptions of integrated circuits into the physical layout level topological mask data required for actually producing the chip. Although the presently available computer-aided design systems greatly facilitate the design process, the current practice still requires highly skilled VLSI design engineers to create the necessary structural level hardware descriptions.

Only a small number of VLSI designers possess the highly specialized skills needed to create structural level integrated circuit hardware descriptions. Even

with the assistance of available VLSI CAD tools, the design process is time consuming and the probability of error is also high because of human involvements. There is a very significant need for a better and more cost effective way to design custom integrated circuits.

In U.S. application Ser. No. 143,821, filed Jan. 13, 1988, and entitled *Knowledge Based Method and Apparatus for Designing Integrated Circuits Using Functional Specifications*, there is disclosed a computer-aided design system and method which enables a user to define the functional requirements for a desired application specific integrated circuit using an easily understood architecture independent functional level representation, such as a flowchart. From this functional level description, a computer implemented expert system generates the detailed structural level definitions needed for producing the application specific integrated circuit. The structural level definitions include a list of the integrated circuit hardware cells needed to achieve the functional specifications. Also included in the detailed structural definitions are the data paths among the selected hardware cells, a system controller for coordinating the operation of the cells and control paths for the selected integrated circuit cells. The various hardware cells are selected from a cell library of previously designed hardware cells of various functions and technical specifications. From this detailed structural level definition it is possible, using either known manual techniques or existing VLSI CAD layout systems to generate the detailed chip level geometrical information (e.g. mask data) required to produce the particular application specific integrated circuit in chip form.

The system described in the aforementioned copending application provides a very significant advance over the methods previously available for designing application specific integrated circuits and opens the possibility for the design and production of application specific integrated circuits by designers, engineers and technicians who may not possess the specialized expert knowledge of a highly skilled VLSI design engineer.

SUMMARY OF THE INVENTION

The present invention provides an improvement over and an extension to the system and method of the aforementioned copending application. Although the system and method of the aforementioned application provides an excellent means for designing ASICs whose functions are implemented in hardware form, there are some occasions where particular functions of the ASCI would best be implemented by software.

The present invention provides a computer-aided system and method for designing an application specific integrated circuit whose intended function is implemented by both a hardware subsystem including hardware elements on the integrated circuit and by a software subsystem including a general purpose microprocessor. Thus, the resulting ASIC produced by the computer-aided design system and method of the present invention includes, on a single integrated circuit chip, a microprocessor for executing the software instructions of the software subsystem and various integrated circuit hardware cells for performing other functions of the integrated circuit.

The system more particularly utilizes a knowledge based expert system, with a knowledge base extracted from expert ASIC designers with a high level of expertise in system design. The knowledge base contains rules for selecting software subroutines from a software subroutine library of predefined functions or operations, as well as rules for selecting hardware cells from a cell library of predefined cells of various types and functions. An inference engine is provided for selecting appropriate hardware cells or software subroutines from these libraries in accordance with the rules of the knowledge base.

The functional specifications of the desired ASIC are independent of any particular architecture or design style and can be defined in a suitable manner, such as in text form or preferably in a flowchart format. The flowchart is a highly effective means of describing a sequence of logical operations, and is well understood by software and hardware designers of varying levels of expertise and training. The designer, when defining the series of operations which implement the intended function of the application specific integrated circuit, may specify whether a particular operation should be implemented in hardware or in software.

The designer may also specify design constraints for the ASIC, such as annual volume, speed, size, pin count, packaging type, power consumption and thermal stability. The knowledge base contains design style rules for selecting, based upon the specified design constraints, an optimum design style for implementation of the hardware subsystem employing various technologies such as programmable logic device (PLD), gate array, standard cell and macro cell.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention will be better understood by reference to the detailed description which follows, taken in connection with the accompanying drawings, in which

FIG. 1 is a block schematic diagram showing how integrated circuit mask data is created from flowchart descriptions by the system of the present invention;

FIG. 2 is a block schematic diagram showing the various elements of the integrated silicon-software compiler system and method of the present invention;

FIG. 3 is a schematic illustration of the system configuration of an ASIC containing both a microprocessor subsystem and a non-microprocessor hardware subsystem;

FIG. 4 is an illustration of a computer display screen which the designer uses in defining the functional specifications of the integrated circuit;

FIGS. 5 and 6 are flowcharts illustrating the functions to be performed by an ASIC in an illustrative design example of the system and method of the invention; and

FIG. 7 is a schematic illustration similar to FIG. 3, but showing the particular ASIC of the design example.

MORE DETAILED DESCRIPTION OF THE INVENTION

System Overview

The overall system flow in accordance with the present invention is illustrated in FIG. 1. The computer-aided system and method of the present invention is represented by the block 10, and for simplicity is referred to herein by the acronym ISSC (integrated silicon-software compiler). The ISSC 10 receives as its input (which are technology and architecture independent) specifications 11, and which define on a functional or behavioral level the functions which are to be performed by the target application specific integrated circuit (ASIC). As illustrated in FIG. 1, the user enters

the functional specifications of the desired target ASIC into the integrated silicon-software compiler (ISSC) 10 in the form of a flowchart 11. The ISSC 10 then generates a netlist 15 from the flowchart. The netlist 15 includes architecture specific definitions of the hardware cells required to implement the functions of the hardware subsystem, a definition of the microprocessor used to implement the functions of the software subsystem, and definitions of a system controller, data paths and control paths for interconnecting the various components. The netlist 15 can be used as input to any existing VLSI layout and routing tool 16 to create mask data 18 for geometrical layout. The ISSC system 10 also generates a program 19 from the flowchart information for implementing the functions of the software subsystem. The program 19 can be stored in an external or internal memory associated with the microprocessor on the target ASIC.

The ISSC system 10 can be operated on a suitable programmed general purpose digital computer. By way of example, one embodiment of the system is operated in a workstation environment such as Sun3 and VAX-Station-II/GPX running UNIX Operating System and X Window Manager. The software uses C programming language and a data base such as INGRES or Gbase. The human interface is mainly done by the use of a pointing device, such as a mouse, with pop up menus, buttons, and a special purpose command language. The permanent data of the integrated circuit design are stored in a database for easy retrieval and update. Main memory temporarily stores the executable code, design data (flowchart, logic, etc.), data base (cell library), and knowledge base. The CPU performs the main tasks of creating and simulating flowcharts and the automatic synthesis of the design.

The primary elements or modules which comprise the ISSC system are shown in FIG. 2. In the embodiment illustrated and described herein, these elements or modules are in the form of software programs, although persons skilled in the appropriate art will recognize that these elements can be easily embodied in other forms, such as in hardware for example.

Referring more particularly to FIG. 2, it will be seen that the ISSC system 10 includes a module or subsystem 20 called KBSC (Knowledge Based Silicon Compiler). The KBSC subsystem 20 corresponds to the system and method described in copending U.S. application Ser. No. 143,821, filed Jan. 13, 1988, the subject matter of which is incorporated herein by reference. As shown in FIG. 2, the KBSC subsystem 20 includes a number of modules or programs which collectively provide an interface with the user for receiving input of the functional specifications for the particular target ASIC and which provide as output a netlist 15 and a program 19. Other major subsystems or modules of the ISSC 10 include a front-end expert system interface (FEXI) 40, a database 60 and a software compiler (SOFTCOM) 70.

The KBSC subsystem 20 includes a program 21 called EDSIM (EDitor SIMulator), which comprises a flowchart editor for creating and editing flowcharts and a flowchart simulator for simulation and verification of flowcharts. The output of EDSIM 21 is an intermediate file 22, referred to as an Antecedent-Action-Form (AAF), which contains a behavioral description of the system that is to be designed. The AAF file 22 contains information about storage elements which compose the data paths, memory elements and external connections to the system. The AAF file **22** is the input to the BLATH program **24**.

The BLATH (Block Level Aaf To Hardware) program **24** is a knowledge based logic synthesis program which selects optimum hardware cells for the hardware subsystem from the hardware cell library and also selects a microprocessor or CPU megacell for use by the software subsystem. The selection is based upon functional descriptions in the flowchart, as specified by the macros assigned to each action represented in the flowchart. BLATH **24** uses a knowledge base **25** extracted from VLSI design experts to select the hardware cells and microprocessor based upon design constraint rules, design style selection rules, and software/hardware routine selection rules. BLATH **24** selects from a database **60** which includes a hardware cell library **61** of previously designed hardware cells and a software subroutine library **62** of previously designed software subroutines. BLATH **24** also outputs an STF file **26** which is used by a system controller generator CONGEN (CONtroller GENerator) **27** and by the software compiler (SOFTCOM) **70**. The controller generator **27** generates a custom designed system controller for controlling the operation of the hardware cells and coordinating with the microprocessor of the software subsystem. Thus, with a functional flowchart input from EDSIM **21**, BLATH **24** selects a microprocessor and all required hardware cells, generates data and control paths, and generates a netlist **15** describing all of ASIC design information.

Target ASIC Design

FIG. 3 illustrates the system configuration of an application specific integrated circuit (ASIC) designed using the ISSC **10**. It consists of two subsystems, a microprocessor subsystem **80** and a non-microprocessor hardware subsystem **90**. The microprocessor subsystem contains a general purpose microprocessor CPU **81** and random access memory (RAM) **82** to execute the software functions of the ASIC. The software program is stored in RAM memory **82**. The hardware subsystem **90** contains special purpose hardware components or cells which perform the functions of the ASIC. Functions are executed in special hardware components primarily to reduce processing time. The various hardware components **91_1, 91_2 . . . 91_n** are integrated into a subsystem which includes a local bus **92**, a PLA control circuit **93**, a register array **94** and an input/output interface **95**. These two subsystems communicate with each other through a system data bus **83**, a system address bus **84** and a system control bus **85**.

EDSIM

The creation and verification of the flowchart is the first step in the VLSI design methodology. The translation from an algorithm to an equivalent flowchart is performed with the flowchart editor, which is contained within the program EDSIM **21**. The flowchart editor provides a working environment for interactive flowchart editing with a designer friendly interface. A graphical display of the flowchart is provided consisting of boxes, diamonds, and lines. All are drawn on the screen and look like a traditional flowchart. The flowchart editor also provides functions such as loading and saving flowcharts. EDSIM will generate an intermediate file **22** for each flowchart. This file is then used by the BLATH program **24** to generate a netlist **15**.

The main editing functions of the flowchart editor include, create, edit, and delete states, conditions, and transitions. The create operation allows the designer to add a new state, condition, or transition to a flowchart. Edit allows the designer to change the position of a state, condition or transition, and delete allows the designer to remove a state, condition or transition from the current flowchart. States which contain actions are represented by boxes, conditions are represented by diamonds, and transitions are represented by lines with arrows showing the direction of the transition.

Once the states and transitions have been created, the designer assigns operations to each state. These operations are made up of macro functions and arguments. The macro function library **63** in database **60** contains a set of macros defining various operations corresponding to the available actions and conditions which can be specified in a flowchart. During the operation of the EDSIM program **21**, the user assigns to each block in the flowchart a macro selected from the macro library and the associated parameters of the macro, if any. FIG. 4 illustrates a screen provided for the designer to specify a macro function in a state in the flowchart. The designer types a macro name along with its parameters in the field associated with the label "enter macro name". In the righthand portion of the screen a macro list is displayed for the designer to select a macro, which can be selected by pointing and clicking a mouse. Once a macro has been selected by this method, the designer would then specify the parameters of the macro. The designer can also select the desired macro type, e.g. hardware (H) or software (S). He then clicks on one of the two circles, "select cell now" or "cell selected by system". If "select cell now" is selected, then EDSIM **21** creates and sends a query to FEXI **40**. The expert system FEXI then queries the database and retrieves suitable cells (either hardware or software as desired by the user). If neither is selected by the user, then both types of cells are displayed. FEXI displays this cell list to the designer and waits for him to select a cell. The designer can query about the features of the cells. EDSIM then returns to normal flowchart editing. If the designer had selected "cell selected by system" from the above screen, then FEXI is not invoked and normal flowchart editing continues.

A list of basic macro functions available in the macro function library **63** is shown in Table 1.

TABLE 1

id	macro name	para-meters	description
1	ADD	2	B = A + B
2	ADD3	3	C = A + B
3	SUB	2	B = A − B
4	MULT	3	C = A * B
5	DIV	3	C = A/B
6	DEC	1	A = A − 1
7	INCR	1	A = A + 1
8	REG	2	B = A
9	CMP	2	compare A and B and set EQ, LT or GT signals
10	CMP0	1	compare A with 0 and set EQ, LT or GT signals
11	NEGATE	1	A = NOT(A)
12	MOD	3	C = A Modulus B
13	POW	3	C = A B
14	DC2	5	decode A into B, C, D, and E
15	EC2	5	encode A, B, C, and D into E
16	MOVE	2	B = A

FEXI

The front-end expert system interface FEXI **40** is invoked by EDSIM **21** for obtaining user design constraints. FEXI starts an interactive session with the user to determine design constraints for the target ASIC such as the following: speed, die size, number of I/O pins, development time, chip count, desired yield, chip cost, package type, technology, power consumption, production volume, pin count, design style, microprocessor type, etc. Once the user has answered the necessary inquiries regarding the design constraints, FEXI **40** utilizes a knowledge base **41** and an inference engine **42** to determine an optimum design style for the hardware components of the integrated circuit from such choices as PLD (programmable logic device), gate array, standard cell, and macro cell. It then determines the technology to be used, and finally selects a hardware cell library to be used. For the software subsystem, a microprocessor is selected among various established microprocessor designs, such as Intel 8086, Motorola 68000, Intel 80286, etc. The database **60** includes software subroutine libraries for each of the standard microprocessors. Once the microprocessor type has been selected, the appropriate software subroutine library for that microprocessor type is selected. FEXI operates as a separate program which runs in parallel with EDSIM **21**. Although illustrated in FIG. **2** separately, FEXI uses the same inference engine used by BLATH **24**.

AAF

The AAF file **22** (Antecedent-Action-Form) is the input to BLATH **24**. It contains the behavioral description of the system that is to be designed. AAF file **22** contains information about storage elements which compose the data paths, memory elements and external connections to the system. This information is contained in the first section of the file. The rest of the file contains a description of the behavior of the system. The first line of the AAF file **22** contains the name of the system and is described by the following form:

name <name>

where <name> is the name of the system.

Storage elements are described at the top of the AAF file **22**. Storage elements can be data paths, external connections or memory elements. They are defined using the "data path", "data mask", "rom" and "ram" statements. These statements are described by the following forms:

"data path" <list>
"data mask" <list>
"rom" <list>
"ram" <list>

where <list> is a list of definitions. These lists are further described below.

Data Paths are defined in the "data path" statement. Data paths are described by the following form:

<pathname>'<'<startbit>':'<stopbit>'>'

<Pathname> is the name of the data path. <Startbit> and <stopbit> are the beginning and ending bit positions of the path. Data Paths must be defined before they can be used by the rest of the AAF.

External Connections behave just like data paths except that they define an external connection to the system. An external connection is described just like a data path in the "data path" statement, except that it is

preceded by an "⊥" to indicate that it is an external connection.

Data mask provide a means of specifying a partial bitfield that is to be treated as another data path. Data mask are described by the following form:

<maskname>'<'<startbit>':'<stopbit>'>''='
  <pathname>'<'<startbit>':'<stopbit>'>'

<Maskname> is the name of the mask that is to be treated as a data path. <Pathname> is the name of the data path that the partial bitfield comes from. <Startbit> and <stopbit> describe the bitrange of the mask and the bitfield of the data path.

Memory elements define RAMs and ROMs to be used in the system. Memories are defined in the "rom" and "ram" statements. Memory elements are described by the following form:

<memname> '[' <startaddr> ':' <stopaddr> ']' <startbit> ':' <stopbit>

where <memname> is the name of the memory. If the memory is a rom, then this name is also the name of a file containing the contents of the rom. <Startaddr> and <stopaddr> are the beginning and ending addresses of the memory. <Startbit> and <stopbit> are the beginning and ending bit positions of the word contained in the memory.

An example is given to illustrate the top part of an AAF file.

name	cpuexample;
data path	acc<0:15>, ir<0:15>, mar<0:15>, iar<0:15>, mdata<0:15>, sum<0:15>;
data mask	adr<0:9> = ir<0:9>;
rom	mem[0:65535]<0:7>;

The behavior of the system is described by state transition information and actions. The state transition information describes the control flow of the system. The actions describe what happens at each state. The representation of the state transition and action information is described in the following sub systems.

State transitions are defined as either being direct or conditional. Direct transitions are described by the following form:

<state1>:<state2>

where <state1> and <state2> are two state names. Direct transitions are an unconditional transition from state1 to state2. This transition will always occur when the system is in state1.

Conditional transitions are described by the following form:

<state1>:.<condition> <state2>

where <state1> and <state2> are the names of two states. <Condition> is the condition that must evaluate to true in order for this transition to be made. The condition is described as an AND function where signals and their compliments are ANDed. Compliments are shown by a "!" and ANDing is described by "*". Some sample conditions are shown below.

a*!b
!GT*!EQ
dog*cat*!bird

State actions consist of macros and assertions. Macros are described by the following form:

<macroname>'.'<instance>'(' <paramlist> ')'

<Macroname> is the name of the macro such as ADD or MOVE. <Instance> is a unique number for each instance of a macro. This allows distinctions to be made

between the same macro at different states. <Paramlist> is a list of zero or more parameter names separated by commas. Macros describe the actions that are to be taken when the system is at a given state. There are four types of parameters that can be used in a macro: Bus, Control, Signal, and Memory.

Bus parameters refer to the data paths or data mask that the macro is applied to. The data path or data mask must be defined in the appropriate statement at the top of the file. Optionally, a bitfield may be specified in the parameter list that will be used instead of the default of using the entire bitrange of the data path.

Control parameters indicate control signals that are necessary for the macro's actions. Controls are defined in the database definition of a macro and used by the rules.

Signal parameters indicate outputs from the macro that go to the controller to indicate the results of the macro. Signals are also defined in the database definition of a macro. Memory parameters refer to a memory that has been previously defined.

An example of the second part of an AAF is given. This example together with the previous example forms a complete AAF.

```
{
start:       ads;
ads  :       ift;
ift  :       dec;
dec  :       excstart;
excend:      end;
end  :       start;
lda  :       excend;
sta  :       excend;
add  :       excend;
bra  :       excend;
brp  :       excend;
excstart .   :.    z2 lda;
excstart     :.    !z2*z3 sta;
excstart     :.    !z2*!z3*z4 add;
excstart     :.    !z2*!z3*!z4*z5*!z0 bra;
excstart     :.    !z2*!z3*!z4*z5*z0 brp;
ads    ::    MOVE.1( iar, mar );
ads    ::    INCR.1( iar );
ift    ::    STORE.1( mem, mar, mdata );
ift    ::    MOVE.2( mdata, ir );
dec    ::    MOVE.3( adr, mar );
dec    ::    DECODE.1( ir<10:15> );
lda    ::    STORE.2( mem, mar, mdata );
lda    ::    MOVE.4( mdata, acc );
sta    ::    LOAD.1( mem, mar, acc );
add    ::    STORE.3( mem, mar, mdata );
add    ::    ADD3.1( mdata, acc, sum );
add    ::    MOVE.5( sum, acc );
bra    ::    MOVE.6( ir<0:9>, iar );
brp    ::    MOVE.7( ir<0:9>, iar );
}
```

BLATH

To design a VLSI system from a flowchart description of a user application, it is necessary to match the functions in a flowchart with hardware cells or software subroutines from the hardware cell library **61** or the software subroutine library **62** (FIG. **2**). This mapping preferably utilizes artificial intelligence techniques since the selection process is complicated and is done on the basis of a number of design parameters and constraints. The concept used for selection is analogous to that used in software compilation. In software compilation a number of subroutines are linked from libraries. In the design of VLSI systems, a functional macro can be mapped to members in the hardware cell library or software subroutine library. BLATH uses a rule based

expert system to select the appropriate hardware cell or software subroutine to perform each action. If the hardware cell library has a number of cells with different geometries for performing the operation specified by the macro, then an appropriate cell can be selected on the basis of factors such as cell function, process technology used, time delay, power consumption, etc.

The knowledge base of BLATH contains information (rules) for:

1) selection of macros
2) merging of two macros
3) mapping of macros to cells
4) merging two cells
5) error diagnostics

The above information is stored in the knowledge base as rules. The first step of cell list generation is the transformation of the flowchart description into a block list. The block list contains a list of the functional blocks to be used in the integrated circuit. The BLATH maps the blocks to cells selected from the cell library, selecting an optimum cell for a block through the use of rules in the knowledge base. The rules which are applied at this point accomplish the following:

Map arguments to data paths
Map actions to macros
Connect these blocks

The rules used by BLATH have the following format: rule name

```
          if
                    ( condition 1 )
                    ( condition 2 )
                      ..
                      ..
                    ( condition n )
          then
                    ( action 1 )
                    ( action 2 )
                      ..
                      ..
                    ( action m ).
```

Exemplary of the rules used by BLATH are the following:

```
Rule 1
          IF      no blocks exist
          THEN
                  generate a system controller.
Rule 2
          IF      a state exists which has a macro AND
                  this macro has not been mapped to a block
          THEN
                  find a corresponding macro in the library
                  and generate a block for this macro.
Rule 3
          IF      there is a transition between two states AND
                  there are macros in these states using the same
                  argument
          THEN
                  make a connection from a register corre-
                  sponding to the first macro to another register
                  corresponding to the second macro.
Rule 4
          IF      a register has only a single connection from
                  another register
          THEN
                  combine these registers into a single register.
Rule 5
          IF      there are two comparators AND
                  input data widths are of the same size AND
                  one input of these is same AND
                  the outputs of the comparators are used to
```

-continued

.		perform the same operation.
	THEN	combine these comparators into a single comparator.
Rule 6		
	IF	there is a data without a register
	THEN	allocate a register for this data.
Rule 7		
	IF	all the blocks have been interconnected AND a block has a few terminals not connected
	THEN	remove the block and its terminals, or issue an error message.
Rule 8		
	IF	memory is to be used, but a block has not been created for it
	THEN	create a memory block with data, address, read and write data and control terminals.
Rule 9		
	IF	a register has a single connection to a counter
	THEN	combine the register and the counter; remove the register and its terminals.
Rule 10		
	IF	there are connections to a terminal of a block from many different blocks
	THEN	insert a multiplexor; remove the connections to the terminals and connect them to the input of the multiplexor; connect the output of the multiplexor to the input of the block.

Additional rules address the following points:

remove cell(s) that can be replaced by using the outputs of other cell(s)

reduce multiplexor trees

use fan-out from the cells, etc.

Database

The database **60** stores information relating to hardware cells, software subroutines, macro functions, user information, and the like. A database interface **65** is provided to allow the system manager to make additions, deletions and modifications to the database. A user table **66** maintains information for every valid user of the ISSC system, including the user identification name, number and password. In the hardware cell library **61** various types of data are stored for each cell, including:

1) functional level information: description of the cell at the register transfer level

2) logic level information: description and terms of flip flops and gates

3) circuit level information: description at the transistor level

4) layout level information: geometrical mask level specifications

A cell table within the hardware cell library **61** contains a record for each cell in the cell library. Every cell in the cell table can be used as instance cell of a larger cell. Once a cell becomes an instance of a larger cell, a data dependency is created. Once this occurs, the contents in the cell can not be altered until the data dependency is eliminated. This information is kept in the attribute "times_used". The attributes of a cell as kept in the cell table are summarized in Table 2:

TABLE 2

Key	Name	Type	Integrity	Description
1	cell_id	i4	>=0	cell identification number
	cell_name	c12	alpha-numeric	cell name assigned by designer
	user_id		i4>=0	user id of the owner of cell
	width		i4>0	width of cell in centi-microns
	height		i4>0	height of cell in centi-microns
	cif_name	c20	unix file name	name of the cif file
	technology_id	i4	>=0	id of associated technology
	test_id		i4>=0	id of the test tool(s) used
	macro_id	i4	>=0	id of macro function for cell
	delay		f4>0	max. prop. delay in nano second
	power		i4>0	power consumption in micro watt
	net_name	c20	unix file name	name of the netlist file
	protection	i4	>=0	flags for user, group, world prot.
	times-used	i4	>=0	counts cell reference quantity
	date	date		date created

An example of a cell is shown below:

cell_id	23
cell_name	XIN02
user_id	8946
width	68
height	2.2
cif_name	xin02.cif
technology_id	1
test_id	1
macro_id	4
delay	0.3
power	25
net_name	xin02.mdl
protection	555
date	88-11-10

Each terminal for a cell is kept in a terminal table. Terminals can be signal, power or ground. The cell_id determines which cell the terminal belongs to. In order to uniquely identify the terminal, both cell_id and terminal_id are required. More than one terminal per cell can have the same terminal name but they are distinguished by the conjunction of cell_id and terminal_id.

The attributes of the terminal table are set forth in Table 3.

TABLE 3

Key	Name	Type	Integrity	Description
1	cell_id	i4	>=0	cell identification number
2	terminal_id	i4	>=0	terminal identification number
	name	c20	alpha-numeric	terminal name
	type	c5	alpha-numeric	terminal type (vdd, gnd, signal)
	layer	c5	alpha-numeric	terminal layer (m1, poly, diff)
	direction	c1	l,r,t,b	terminal exit side
	x	i4		x coordinate of the terminal
	y	i4		y coordinate of the terminal
	width	i4		terminal width

The cellref table contains the hierarchical cell structure. Using this table, it is possible to learn if a cell has any parent, and therefore a data dependency. The attributes of the cellref table are set forth in Table 4:

TABLE 4

Key	Name	Type	Integrity	Description
1	parent_id	i4	>=0	cell id of the parent
2	child_id	i4	>=0	cell id of the child
3	instance_id	i4	>=0	instance id to distinguish cells
	x	i4		x coordinate of the instance
	y	i4		y coordinate of the instance
	MX	i2	Boolean	True if inst. is mirrored in X
	MY	i2	Boolean	True if inst. is mirrored in Y
	RO	i2	Boolean	Rotation variable in X
	R1	i2	Boolean	Rotation variable in Y

Technology refers to a particular process run of a IC foundry. For each entry in the technology table, there is a set of cells that uses the technology (called cell library). Each record in this table is a unique process run to not only within the foundry, but also within the corporation. The attributes of the technology table are set forth in Table 5 below:

TABLE 5

Key	Name	Type	Integrity	Description
1	technology_id	i4	>=0	id that identifies technology
	facility	c20	alpha-numeric	name and location of fab house
	device	c10	alpha-numeric	CMOS, bipolar, HCMOS, GaAs, etc.
	feature_size	i4	>0	channel width of a transistor
	no_metal	i2	>=0	number of metals used
	no_poly	i2	>=0	number of polysilicons used
	well_type	c10	alpha-numeric	N-Well, P-Well, etc.

Every standard frame used for a specific technology is described in the frame table. Frame table is related to the technology table by the key technology_id. Each frame must belong to one and only one technology, while each technology can have many frames. Each record in the frame table represents a frame. The attributes of the frame table are set forth in Table 6 below:

TABLE 6

Key	Name	Type	Integrity	Description
1	frame_id	i4	>=0	id that identifies each frame
2	technology_id	i4	>=0	id of corresponding technology
	no_pads	i2	>0	number of pads in the frame
	in_width	i4	>=0	width of usable space in frame
	in_height	i4	>=0	height of usable space in frame
	out_width	i4	>0	total frame width with pads
	out_height	i4	>0	total frame height with pads

The software subroutine library 62 includes for each microprocessor type, predefined software subroutines which correspond to the macro functions which can be implemented in software. Typical subroutines used in ISSC are standard I/O macros, data conversion macros,

character conversion macros, and string manipulation macros.

Every macro function used in categorizing a cell in terms of its function is stored as a single record entry in a macro table contained within the macro function library 63. Many cells within the cell library can be capable of performing the function of a macro. Each macro in the table is uniquely identified by the key macro_id. The attributes of the macro table are set forth in Table 7:

TABLE 7

Key	Name	Type	Integrity	Description
1	macro_id	i4	>=0	id that identifies each macro
	macro_name	c20	alpha-numeric	macro name describing function
	no_inputs	i2	>=0	number of inputs in function
	no_outputs	i2	>=0	number of outputs in function
	description	c80	sdl	description

CONGEN

CONGEN 27 used the STF file from BLATH 24 to design a PLA based system controller for the chip which generates control signals to enable the respective hardware cells or the microprocessor. The STF file is translated to Boolean equations for PLA logic that implements next state codes and output functions. A unique code is assigned for each state by applying heuristic rules for state assignment. The output of CONGEN is an input parameter file for automatic PLA layout generation.

SOFTCOM

SOFTCOM 70 takes as input the STF file 26 created by BLATH 24 and generates an assembly program which is the monitor program for the microprocessor. Some of the variables of this program are:

NUM_SHARED is the number of shared variables between software and hardware

REG_ARRAY[0.. NUM_SHARED] is a external register array.

NUM_SUB is the number of software routines in the system.

SHARED_VARS is an array of memory locations at which the shared variables are stored.

change_flag: associated with each shared variable is a flag which is set if that variable's value is changed.

The program format is as follows:

```
1)  set change_flag for shared variables to false.
2)  get subroutine code from REG_ARRAY[0]
3)  if code == 1 then call subroutine 1
    if code == 2 then call subroutine 2
    . . .
    . . .
    if code == NUM_SUB then call subroutine_num_sub
4)  for i = 1 to NUM_SHARED do
    {
        if change_flag set then
        move SHARED_VARS[i] TO REG_ARRAY[i]
    }
5)  transfer control to the system controller
```

Design Example

This example illustrates the design approach employed in designing an application specific integrated circuit which does simple computer graphics operations such as translation, mirror image of X axis or Y axis, and rotation. All of those operations are involved in matrix multiplication with the transformation axis. The matrix multiplication will be implemented in hardware to speed up the process. FIG. **5** is a flowchart illustrating a first software portion (software portion #**1**) of the foregoing program which is to be processed by the microprocessor subsystem within the target ASIC. FIG. **6** is a continuation of the flowchart of FIG. **5** and illustrates the remaining portion of software portion #**1** as well as software portion #**2** to be processed by the microprocessor subsystem within the target ASIC, and additionally shows the non-microprocessor hardware subsystem within the target ASIC.

FIG. **7** is a block schematic diagram similar to the general diagram of FIG. **3** but showing the components which would be utilized in the above-noted specific example. To avoid repetitive description, elements in FIG. **7** which correspond to elements previously described in connection with FIG. **3** will be identified by the same reference characters with prime notation (′) added. The microprocessor subsystem **80**′ contains CPU **81**′ and RAM **82**′ (main memory). The execution code of software portions #**1** and #**2** (**82***a*, **82***b*) are stored in the main memory of microprocessor subsystem. The data segment **82***c* contains the data to be processed by both CPU and non-microprocessor hardware subsystem. The data in data segment **82***c* are sent to data registers via the system data bus **83**′. The communication between microprocessor subsystem **80**′ and hardware subsystem **90**′ is coordinated by using interrupts. After software portion #**1** (**82***a*) is executed, microprocessor subsystem **80**′ reads the status registers to see if hardware subsystem **90**′ is busy. If hardware subsystem **90**′ is idle, its status registers are set, and the hardware subsystem **90**′ is activated.

When hardware subsystem **90**′ finishes the task, it generates an interrupt signal to microprocessor subsystem **80**′. The interrupt is serviced by interrupt handler **82***d*. The main tasks of the interrupt handler include:

push contents of microprocessor subsystem **80**′ registers into stack.

transfer data from hardware subsystem **90**′ to microprocessor subsystem **80**′.

reset the status register of hardware subsystem **90**′.

restore the contents of microprocessor subsystem **80**′ registers from stack.

continue the execution of software portion #**2** (**82***b*).

The execution flow of the entire flowchart (including software and hardware portions) is as follows:

1. Microprocessor subsystem **80**′ executes the codes in software portion #(**82***a*).
2. Microprocessor subsystem **80**′ activates hardware subsystem **80**′ by overwriting the status registers, then sends the data to data registers.
3. Hardware subsystem **90**′ processes the data in data registers
4. After hardware subsystem **90**′ processed the data, it sends interrupt signal to microprocessor subsystem **80**′.
5. The interrupt handler **82***d* of microprocessor subsystem **80**′ handles the interrupt, transfers the data from data registers back to microprocessor subsystem, and resets the status registers. After the interrupt is processed, the execution flow is returned to software portion #**2**.
6. Microprocessor subsystem **80**′ executes the codes in software portion #**2**.
7. Done.

As shown in FIG. **6**, states **24–35** and conditions **9–11** are implemented in hardware subsystem to perform multiply/accumulate operation. A multiplier and an adder are used in state **28** to implement the multiply/accumulate operations. The blocks indicated as state **24**, state **25**, state **27**, state **29**, state **32**, and state **34** use counters i, j, k, which have increment and clear capabilities. In state **26** and state **31**, temp is used as a temporary register to store the intermediate result. State **30**, state **33**, and state **35** use a comparator. The result of the comparison will affect the control flow which specified in cond**9**, cond**10**, and cond**11**.

The execution flow of hardware subsystem is as follows:

1. Microprocessor subsystem sends a start signal to hardware subsystem via system control bus.
2. All the data are loaded to data registers via system data bus.
3. Loop counters i, j, and k are reset to zero initially. Register temp is also reset to zero.
4. Multiply/Accumulate operation.
5. Increment the loop counters and check if the end of the loops. If it is not end of the loops, go to step 3; else exits the loops.
6. The result of multiply/accumulate operation is stored in the output register that is connected to system data bus.
7. An interrupt signal is sent to the microprocessor subsystem via system control bus.

While the invention has been described herein with reference to a specific embodiment, the description is illustrative of the invention and is not to be construed as limiting the invention. Various modifications and applications may occur to those skilled in the art without departing from the true spirt and scope of the invention as defined by the appended claims.

That which is claimed is:

1. A computer-aided design system for designing an application specific integrated circuit from architecture independent functional specifications for the integrated circuit, whose intended function is implemented by both a hardware subsystem including hardware elments on the integrated circuit and by a software subsystem including software instructions executed by a microprocessor on the integrated circuit, comprising

input means operable by a user for defining architecture independent functional specifications for the application specific integrated circuit wherein at least one of said architecture independent functional specifications is free of indication that an intended function of said integrated circuit is implanted on said hardware subsystem or said software subsystem, and

computer operated means for translating the architecture independent functional specifications into an architecture specific structural level definition of the integrated circuit, said computer operated translating means comprising

means defining software instructions of the software subsystem from the architecture independent functional specifications, at least one of which is free of indication that an intended function of said integrated circuit is implemented on said hardware subsystem or said software subsystem,

means defining a microprocessor for executing the software instructions of the software subsystem from the architecture independent functional specifications, at least one of which is free of indication that an intended function of said integrated circuit is implemented on said hardware subsystem or said software subsystem,

means defining hardware elements for executing the hardware functions of the hardware subsystem from the architecture independent functional specifications, at least one of which is free of indication that an intended function of said integrated circuit is implemented on said hardware subsystem or said software subsystem, and

means defining interconnections between the microprocessor for executing the software instructions of the software subsystem and the hardware elements of the hardware subsystem.

2. The system as defined in claim 1 wherein said architecture independent functional specifications are comprised of a series of operations, and wherein said computer operated means comprises

a cell library defining a set of available integrated circuit hardware cells for performing said operations;

a software subroutine library defining a set of available software subroutines for performing said operations; and

selection means for selecting from said cell library or from said software subroutine library for each operation specified by said specification input means, appropriate hardware cells or software subroutines for performing the operations specified by the functional specifications.

3. The system as defined in claim 2 wherein said selection means comprises an expert system including a knowledge base containing rules for selecting hardware cells from said cell library and software subroutines from said software subroutine library, and inference engine means for selecting appropriate hardware cells or software subroutines from said libraries in accordance with the rules of said knowledge base.

4. A computer-aided design system for designing an application specific integrated circuit from architecture independent functional specifications for the integrated circuit whose intended function is implemented by both a hardware subsystem including hardware elements on the integrated circuit and by a software subsystem including software instructions executed by a microprocessor on the integrated circuit, comprising

a macro library defining a set of architecture independent operations comprised of actions and conditions wherein at least one of said architecture independent operations is free of indication that the operation is implemented on said hardware subsystem or said software subsystem;

specification input means operable by a user for defining architecture independent functional specifications for the integrated circuit wherein at least one of said architecture independent functional specifications is free of indication that an intended function of said integrated circuit is implemented on said hardware subsystem or said software subsystem, said architecture independent functional specifications being comprised of a series of architecture independent operations which implement the intended function of the application specific integrated circuit wherein at least one of said operations is free of indication that an intended function of the integrated circuit is implemented or said hardware subsystem or said software subsystem, said specification input means including means to permit the user to specify for each operation in the defined series of operations a macro selected from said macro library;

a hardware cell library defining a set of available integrated circuit hardware cells for performing operations defined in said macro library wherein at least one of said operations is free of indication that an intended function of the integrated circuit is implemented on said hardware subsystem or said software subsystem;

a software subroutine library defining a set of available software subroutines for performing operations defined in said macro library wherein at least one of said operations is free of indication that an intended function of the integrated circuit is implemented on said hardware subsystem or said software subsystem;

selection means for selecting from said hardware cell library or from said software subroutine library for each macro specified by said specification input means, appropriate hardware cells or software subroutines for performing the operation defined by the specified macro wherein at least one of said operations is free of indication that an intended function of the integrated circuit is implemented on said hardware subsystem or said software subsystem, said selection means comprising an expert system including a knowledge base containing rules for selecting hardware cells from said hardware cell library and software subroutines fiom said software subroutine library, and inference engine means for selecting appropriate hardware cells or software subroutines from said libraries in accordance with the rules of said knowledge base.

5. The system as defined in claim 4 wherein said specification input means includes means to enable the user, when specifying a macro selected from said macro library, to define whether the macro function is to be implemented in hardware or in software.

6. The system as defined in claim 4 additionally including design constraint input means operable by a user for defining design constraints for the application specific integrated circuit.

7. The system as defined in claim 6 wherein said design constraint input means includes means to enable the user to define one or more design constraints for the application specific integrated circuit selected from the group consisting of annual volume, speed, size, pin count, packaging type, power consumption, and thermal stability.

8. The system as defined in claim 7 wherein said knowledge base additionally contains design style rules for selecting, based upon the design constraints, an optimum design style for implementation of the hardware subsystem.

9. The system as defined in claim 8 wherein said design style is selected from the group consisting of programmable logic device (PLD), gate array, standard cell, and macro cell.

10. The system as defined in claim 6 wherein said design constraint input means includes means to enable the user to select, from a set of predefined available

standard microprocessors, the particular microprocessor type to be utilized by the software subsystem.

11. The system as defined in claim 10 wherein said software subroutine library includes sets of software subroutines corresponding to each available standard microprocessor type.

12. The system as defined in claim 6 wherein said knowledge base additionally contains design constraint rules for selecting appropriate hardware cells or software subroutines based upon the design constraints defined by said design constraint input means.

13. The system as defined in claim 4 wherein said specification input means comprises flowchart editor means for creating a flowchart having elements representing said series of actions and conditions.

14. The system as defined in claim 4 wherein said specification input means comprises means for receiving user input of a list defining the series of actions and conditions which implement the function of the application specific integrated circuit.

15. The system as defined in claim 4 wherein said selection means comprises means for generating a netlist defining the microprocessor to be utilized by the software subsystem and the hardware cells to be utilized by the hardware subsystem, and interconnection information for interconnecting the microprocessor and the hardware cells to perform the intended function of the application specific integrated circuit.

16. The system as defined in claim 15 wherein said selection means additionally includes control generator means for generating a controller and control paths for controlling operation of the microprocessor and the hardware cells selected by said cell selection means.

17. The system as defined in claim 15 additionally including mask data generator means for generating from said netlist the mask data required to produce an integrated circuit having the specified functional requirements.

18. The system as defined in claim 4 additionally including software compiler means for generating software operable by the microprocessor for performing functions of the software subsystem.

19. A computer-aided design system for designing an application specific integrated circuit from a flowchart defining architecture independent functional requirements of the integrated circuit, whose intended function is implemented by both a hardware subsystem including hardware elements on the integrated circuit and by a software subsystem including software instructions executed by a microprocessor on the integrated circuit, comprising

a computer system;

a macro library, stored in said computer system, defining a set of possible architecture independent actions and conditions wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem;

flowchart editor means, executing on said computer system, operable by a user for creating a flowchart having elements representing said architecture independent actions and conditions wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem, said flowchart editor means, executing on said computer system,

including macro specification means, executing on said computer system, for permitting the user to specify for each architecture independent action or condition represented in the flowchart a macro selected from said macro library;

a hardware cell library, stored in said computer system, defining a set of available integrated circuit hardware cells for performing the architecture independent actions and conditions defined in the macro library wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem;

a software subroutine library, stored in said computer system, defining a set of available software subroutines for performing the architecture independent actions and conditions defined in the macro library wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem; and

selection means, executing on said computer system, for selecting from said hardware cell library or from said software subroutine library for each macro specified by said flowchart editor means, executing on said computer system, appropriate hardware cells or software subroutines for performing the architecture independent action or condition defined by the specified macro, and for generating a netlist defining the microprocessor to be utilized by the software subsystem and the hardware cells to be utilized by the hardware subsystem, and interconnection information for interconnecting the microprocessor for executing the software subroutines and the hardware cells to perform the intended function of the application specific integrated circuit.

20. The system as defined in claim 19 wherein said selection means comprises a knowledge base containing rules for selecting hardware cells from said cell library and software subroutines from said software subroutine library, and inference engine means for selecting appropriate hardware cells or software subroutines from said libraries in accordance with the rules of said knowledge base.

21. A computer-aided design process for designing, in a computer system, an application specific integrated circuit from architecture independent functional specifications for the integrated circuit, whose intended function is implemented by both a hardware subsystem including hardware elements on the integrated circuit and by a software subsystem including software instructions executed by a microprocessor on the integrated circuit, comprising the following steps performed by said computer system:

storing in a macro library in said computer system, a set of definitions of possible architecture independent actions and conditions wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem;

storing in a hardware cell library in said computer system, data describing a set of available integrated circuit hardware cells for performing the architecture independent actions and conditions wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem;

storing in a software subroutine library in said computer system, data describing a set of software subroutines for performing the architecture independent actions and conditions wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem;

storing in an expert system knowledge base in said computer system, a set of rules for selecting hardware cells or software subroutines to perform the architecture independent actions and conditions wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem;

describing for a proposed application specific integrated circuit a series of architecture independent actions and conditions which will implement the intended function of the application specific integrated circuit wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem;

specifying for each described architecture independent action and condition of the series one of said stored definitions from the macro library stored in said computer system which corresponds to the desired architecture independent action or condition to be performed; and

selecting from said hardware cell library stored in said computer system or from said software subroutine library stored in said computer system for each of the specified definitions, appropriate integrated circuit hardware cells of software subroutines for performing the desired function of the application specific integrated circuit.

22. A process as defined in claim **21** wherein said step of selecting a hardware cell or a software subroutine comprises applying to the specified definition of the action or condition to be performed, a set of cell selection rules and software subroutine selection rules stored in the knowledge base in said computer system.

23. A process as defined in claim **21** including the further step performed by said computer system of generating a netlist defining the microprocessor and the hardware cells which are needed to perform the desired function of the integrated circuit and the connections therebetween.

24. A process as defined in claim **23** including the further step of generating from said netlist the mask data required to produce an integrated circuit having the specified functional requirements.

25. A computer-aided knowledge based design process for designing in a computer system an application specific integrated circuit from architecture independent functional specifications for the integrated circuit, whose intended function is implemented by both a hardware subsystem including hardware elements on the integrated circuit and by a software subsystem including software instructions executed by a microprocessor on the integrated circuit, comprising the following steps performed by said computer system:

storing in a macro library in said computer system, a set of macros defining possible architecture independent actions and conditions wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem;

storing in a hardware cell library in said computer system, data describing a set of available integrated circuit hardware cells for performing the architecture independent actions and conditions wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem;

storing in a software subroutine library in said computer system, data describing a set of available software subroutines for performing the architecture independent actions and conditions wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem;

storing in a knowledge base in said computer system, a set of rules for selecting hardware cells from said hardware cell library and software subroutines from said software subroutine library to perform the architecture independent actions and conditions defined by the stored macros wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem;

describing for a proposed application specific integrated circuit a series of architecture independent actions and conditions which carry out the function to be performed by the hardware cells and software subroutines of the integrated circuit wherein at least one of said architecture independent actions and conditions is free of indication that the action or condition is implemented on said hardware subsystem or said software subsystem;

specifying for each described action and condition of said series a macro selected from the macro library stored in said computer system which corresponds to the action or condition; and

applying rules of said knowledge base stored in said computer system to the specified macros to select from said hardware cell library stored in said computer system and from said software subroutine library stored in said computer system, the hardware cells and software subroutines required for performing the desired function of the application specific integrated circuit.

26. A process as defined in claim **25** wherein said step of describing a series of actions and conditions comprises creating a flowchart comprised of elements representing actions and conditions.

27. A process as defined in claim **25** wherein said step of specifying a macro selected from the macro library further includes defining whether the macro function is to be implemented in hardware or in software.

28. A process as defined in claim **27** including the step of generating software code for use by the microprocessor in performing the macro functions which are specified to be implemented in software.

29. A process as defined in claim **25** including defining one or more design constraints for the application specific integrated circuit selected from the group consisting of annual volume, speed, size, pin count, packaging type, power consumption, and thermal stability.

30. A process as defined in claim **29** wherein said knowledge base additionally contains design style rules for selecting, based upon the design constraints, an optimum design style for implementation of the hardware subsystem, and wherein said step of applying the rules of the knowledge base includes selecting, based upon the design constraints, an optimum design style for implementation of the hardware subsystem.

31. A process as defined in claim **30** wherein said design style is selected from the group consisting of programmable logic device (PLD), gate array, standard cell, and macro cell.

32. A process as defined in claim **25** wherein

said step of specifying a macro selected from the macro library further includes indicating, for at least certain ones of the specified macros, that the selection of whether the macro function is to be implemented in hardware or in software is to be made by the computer system, and

wherein said knowledge base additionally contains implementation rules for determining the optimum implementation of macro functions either in the hardware subsystem or in the software subsystem, and

wherein said step of applying the rules of the knowledge base includes determining, based upon the implementation rules in said knowledge base, the optimum implementation of the macro function either in the hardware subsystem or the software subsystem.

* * * * *

Teresa M. Corbin (SBN 132360)
Thomas Mavrakakis (SBN 177927)
HOWREY SIMON ARNOLD & WHITE, LLP
301 Ravenswood Avenue
Menlo Park, California 94025
Telephone: (650) 463-8100
Facsimile: (650) 463-8400

Attorneys for Plaintiff SYNOPSYS, INC.
and for Defendants AEROFLEX INCORPORATED,
AMI SEMICONDUCTOR, INC., MATROX
ELECTRONIC SYSTEMS, LTD., MATROX
GRAPHICS INC., MATROX INTERNATIONAL
CORP. and MATROX TECH, INC.

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

SAN FRANCISCO DIVISION

RICOH COMPANY, LTD., Plaintiff, vs. AEROFLEX INCORPORATED, et al., Defendants.	Case No. C03-04669 MJJ (EMC) Case No. C03-2289 MJJ (EMC) **NOTICE OF MANUAL FILING OF EXHIBIT 4 TO RESPONSIVE CLAIM CONSTRUCTION BRIEF**
SYNOPSYS, INC., Plaintiff, vs. RICOH COMPANY, LTD., a Japanese corporation Defendant.	Date: October 29, 2004 Time: 9:30 AM Courtroom: 11 Judge: Martin J. Jenkins

MANUAL FILING NOTIFICATION

Regarding: Exhibit 4 to Responsive Claim Construction Brief

This filing is in paper or physical form only, and is being maintained in the case file in the Clerk's office. If you are a participant in this case, this filingwill be served in hard-copy shortly. For information on retrieving this filing directly from the court, please see the court's main web site at http://www.cand.uscourts.gov under Frequently Asked Questions (FAQ).

This filing was not efiled for the following reason(s):

[X] Voluminous Document (PDF file size larger than the efiling system allows)

[_] Unable to Scan Documents

[_] Physical Object (description): ____

____

[_] Non-Graphic/Text Computer File (audio, video, etc.) on CD or other media

[_] Item Under Seal

[_] Conformance with the Judicial Conference Privacy Policy (General Order 53).

[_] Other (description): ____

____

United States Patent [19]
Darringer et al.

[11] Patent Number: 4,703,435
[45] Date of Patent: Oct. 27, 1987

[54] LOGIC SYNTHESIZER

[75] Inventors: John A. Darringer, Mahopac; William H. Joyner, Jr., Katonah, both of N.Y.

[73] Assignee: International Business Machines Corporation, Armonk, N.Y.

[21] Appl. No.: 631,364

[22] Filed: Jul. 16, 1984

[51] Int. Cl.⁴ G06F 7/00; G06F 15/60
[52] U.S. Cl. 364/489; 364/300; 364/488
[58] Field of Search 364/488, 489, 490, 491, 364/300; 307/303

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,377,849	3/1983	Finger et al.	364/491
4,580,228	4/1986	Noto	364/300 X
4,591,993	5/1986	Griffin et al.	364/491
4,612,618	9/1986	Pryor et al.	364/488 X

FOREIGN PATENT DOCUMENTS

0168650 1/1986 European Pat. Off. .

OTHER PUBLICATIONS

Friedman et al., "Methods used in an Automatic Logic Design Generator (Alert)", *IEEE Trans. Comp.* C-18, pp. 593-614, 1969.
Mitchell et al., "The Use of High Speed Proms to Generate Boolean Functions,", *Wescon Technical Papers,* Sep. 1978, pp. 1-7.
Mano, "Digital Logic and Computer Design", Ch. 3, pp. 72-103, 1979.
Introduction to the Automated Synthesis of Computer; Herbert Schorr; Department of Electrical Engineering Digital Systems.
Laboratory Technical Report No. 16 (Mar. 1962). Ph.D. Thesis, Princeton University.
Minimization of Boolean Functions; F. J. Hill et al.; "Introduction to Switch Theory and Logical Design", 1973.
The Description, Simulation, and Automatic Implementation of Digital Computer Processors; John A. Darringer.
The Experimental Compiling System; F. E. Allen; IBM J. Res. Development; vol. 24, No. 6, Nov. 1980.
Logic Synthesis Through Local Transformation; John A. Darringer; IBM Journal of Research and Development; vol. 25, No. 4, Jul. 1981.
Programming Language; Yaohan Chu; vol. 8, No. 10, 10/65.
Development and Application of a Designer Oriented Cyclic Simulator; G. J. Parasch; 13th DA Conference 1976.
Logic Synthesis; Melvin A. Breuer; M. Breuer "Design Automation of Digital Systems", Prentice-Hall 1972.
Synthesis of Combinational Logic Networks; D. L. Dietmeyer "Logic Design of Digital Systems"; Allyn & Bacon, Boston 1978.
Quality of Designs from an Automatic Logic Generator (Alert)*; Theodore D. Friedman and Sih-Chin Yang; 7th DA Conference 1970.
On Logic Comparison; Leonard Berman; 18th DA Conference 1981.

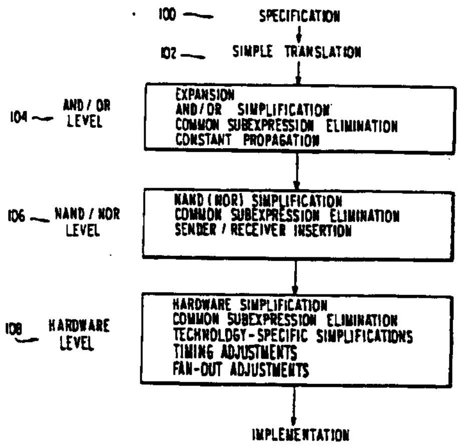
(List continued on next page.)

Primary Examiner—Errol A. Krass
Assistant Examiner—Joseph L. Dixon
Attorney, Agent, or Firm—Sughrue, Mion, Zinn, Macpeak, and Seas

[57] **ABSTRACT**

Logic is synthesized from a flowchart-level description by first generating an AND/OR logic design, simplifying the AND/OR logic, converting the AND/OR logic to NAND or NOR logic, applying particular sequences of simplifying transformations to the NAND or NOR logic, converting the simplified NAND or NOR logic to a target technology, and simplifying the target technology where possible. The end result is an interconnection of primitives of the target technology in a language from which automated logic diagrams can be produced.

24 Claims, 33 Drawing Figures



OTHER PUBLICATIONS

Automated Exploration of the Design Space for Register-Transfer (RT) Systems: Barbacci Mario Roberto;
The Design and Analysis of an Automated Design Style Selector: Thomas Donald Earl, Jr.
Automation of Module Set Independent Register–Transfer Level Design: Edward Alfred Snow, III.
A new Look at Logic Synthesis; John A. Darringer; 17th DA Conference 1980.
Experiments in Logic Synthesis; John A. Darringer; IEEE ICCC 1980.
Methods used in an Automatic Logic Design Generator (Alert); Theodore D. Friedman; IEEE Transactions on Computers; vol. C-18, No. 7, Jul. 1969.
Register-Transfer Level Digital Design Automation: The Allocation Process; Louis Hafer; 15 DA Conference 1978.
The CMU Design Automation System; An Example of Automated Data Path Design; A. Parker et al.; 16 DA Conference 1979.
Lores-Logic Reorganization System; Shunichiro Nakamura et al.; 15 DA Conference 1978.
Translation of a DDL Digital System Specification to Boolean Equations; James R. Duley and Donald L. Dietmeyer, IEEE Trans. on Comp. vol. C-18. No. 14, '69.
DDL-A Digital System Design Language; James Robert Duley; Ph.D. 1967.

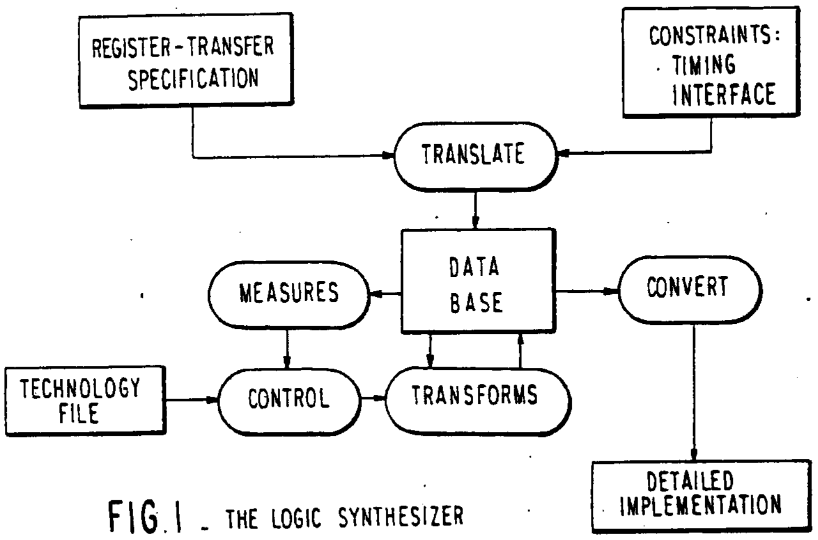
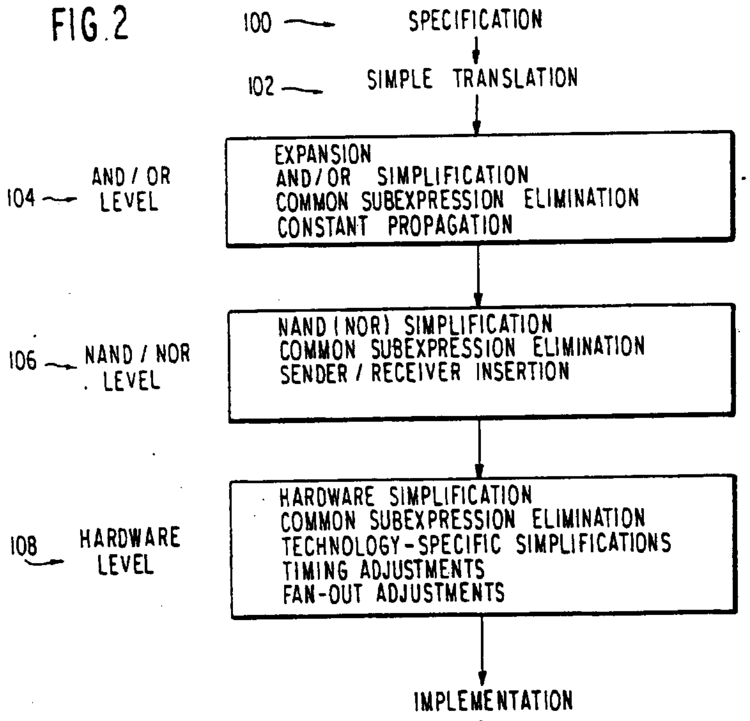


FIG. 1 - THE LOGIC SYNTHESIZER

FIG. 2



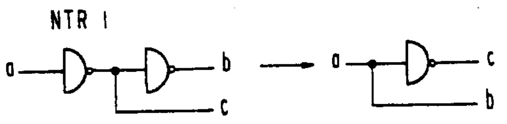


FIG.3a

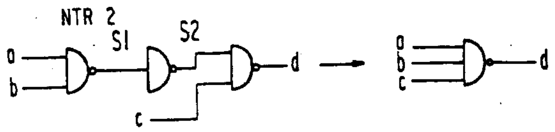


FIG.3b

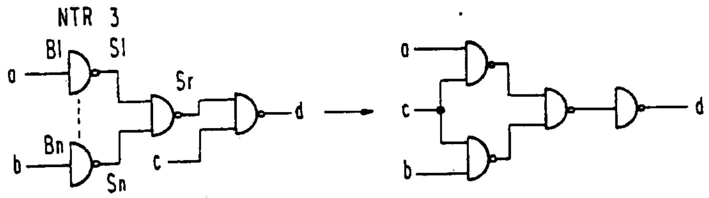


FIG.3c

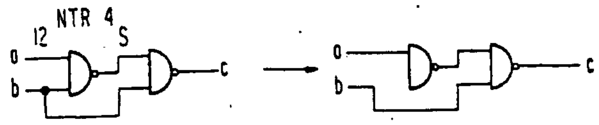


FIG.3d

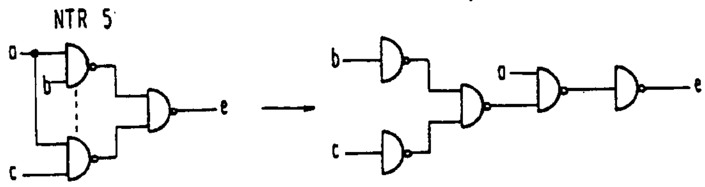


FIG.3e

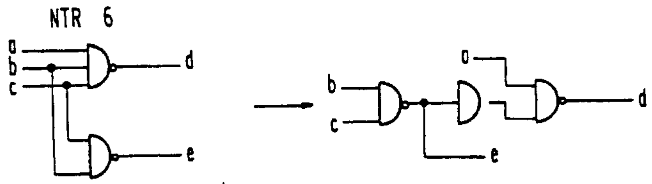


FIG.3f

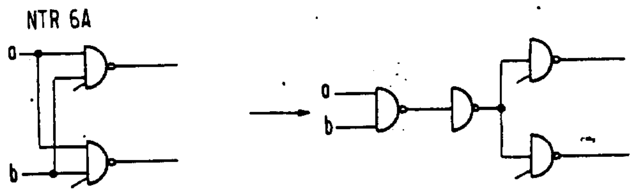


FIG.3g

DEF012506

FIG. 3h

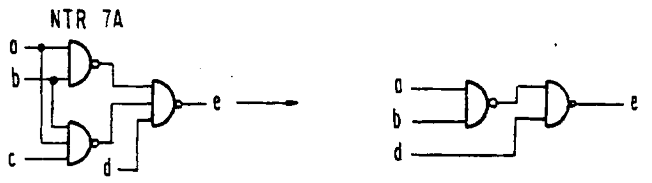


FIG. 3i

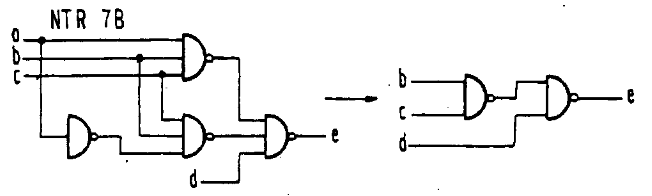


FIG. 3j

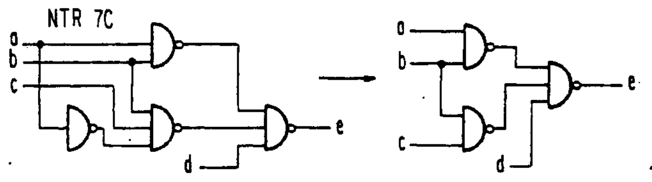


FIG. 3k

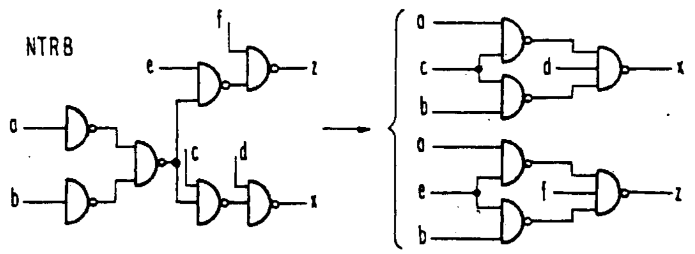


FIG. 3l

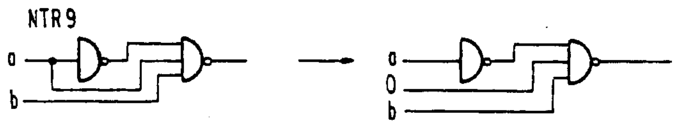


FIG. 3m

FIG. 3n

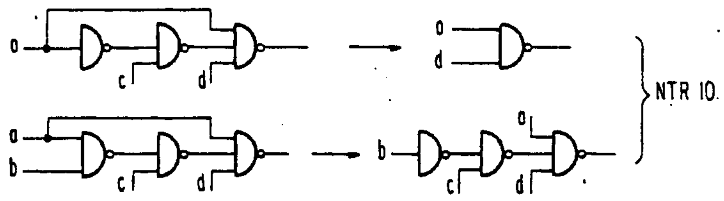


FIG. 3o

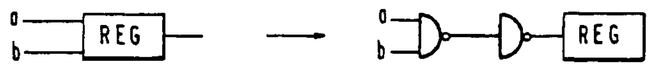


FIG. 3p

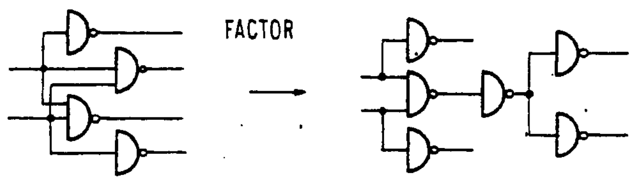


FIG. 4

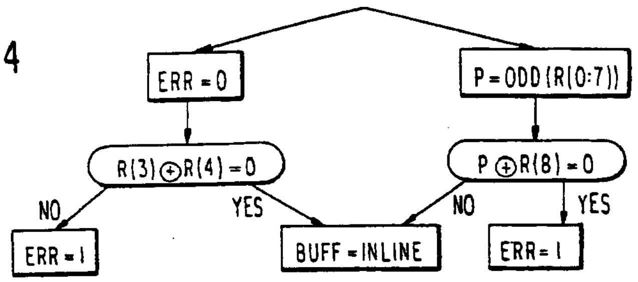


FIG. 5a
ELIMINATE COMMON TERMS

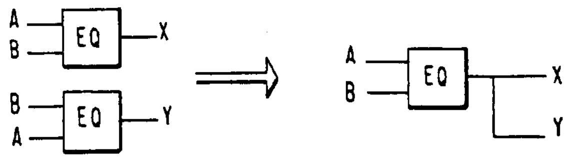


FIG. 5b
SIMPLIFY PARITY OPERATORS

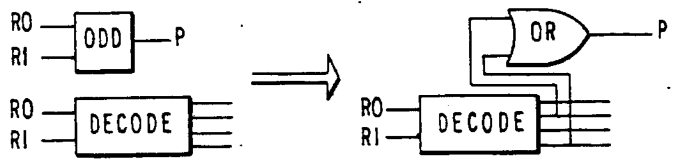


FIG. 6

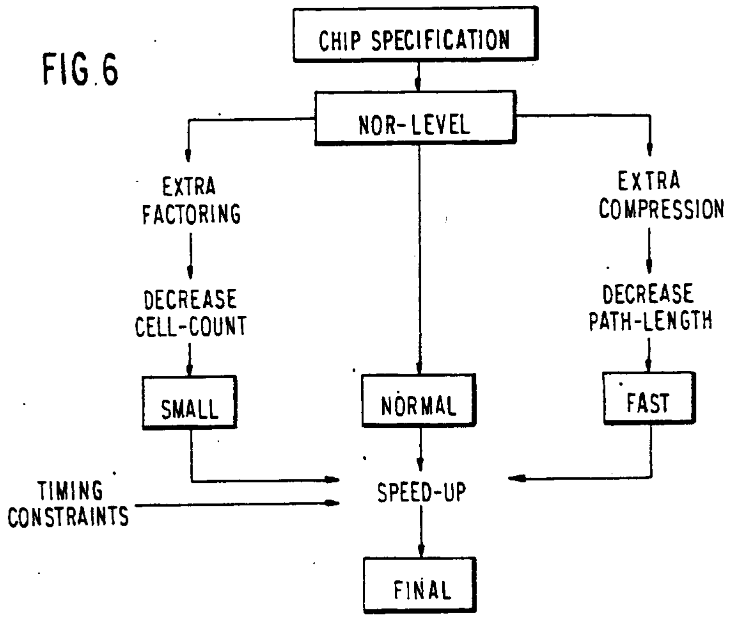


FIG. 7a
MERGE RECEIVERS

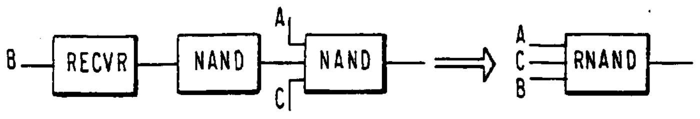


FIG. 7b
INSERT DOTS

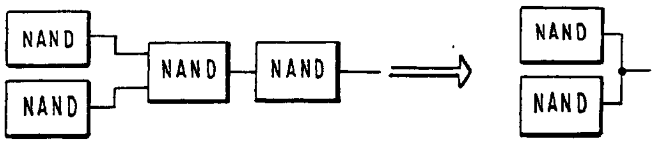


FIG. 7c

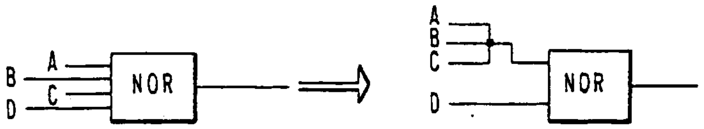


FIG. 8a

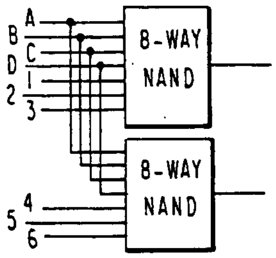


FIG. 8b

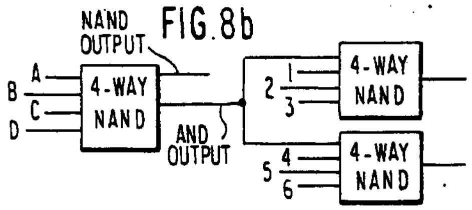


FIG. 9a

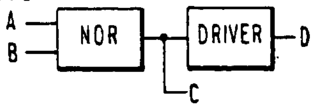


FIG. 9b

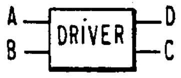
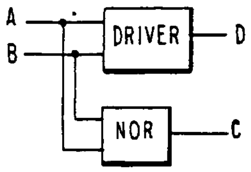


FIG. 9c



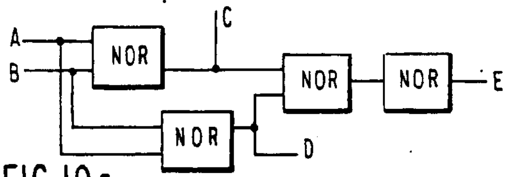


FIG. 10a

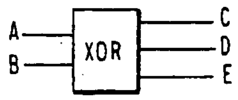


FIG. 10b

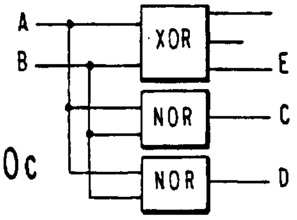


FIG. 10c

LOGIC SYNTHESIZER

BACKGROUND OF THE INVENTION

This invention is directed to logic design, and more particularly to a method of automated logic design.

As the complexity of processors has increased, the task of processor logic design has become more difficult. The designer may begin by designing a flow chart or other register-transfer level description to describe the intended operation of the processor, and the processor operation is then simulated from this description in order to ensure that a processor operating in accordance with the flow chart will provide the desired results. A logic implementation is then designed to achieve the operation described in the flow chart, and the resulting logic diagram and original flow chart specification are compared to ensure consistency. Finally, a physical layout is designed in accordance with the logic diagram implementation.

The above process has become significantly more difficult and extraordinarily time consuming with the increasing complexity of the processors being designed. For example, each chip in the 3081 processor available from International Business Machines Corporation includes over 700 circuits capable of performing extremely complex functions. The flow chart specification of such a processor will be quite complex, and even a first attempt at a logic diagram implementation will require a substantial amount of time. Further, with increasing processor complexity, the competing interests of gate count and timing constraints become increasingly difficult to satisfy. More particularly, a typical timing constraint may be that a signal must be provided from the output of register A to the input of register B within some predetermined period of time, and the designer may first propose a logic arrangement intended to satisfy this timing constraint while using a minimal number of gates in the circuit path between registers A and B. After timing analysis, however, it may be discovered that the timing constraint has not been satisfied, and the designer must then revise the arrangement of logic between the registers A and B, e.g., by using a larger number of gates to improve the processing speed in that area. Several iterations of design may be required before a logic design is obtained which indeed satisfies all timing constraints with the minimum gate count, and it is therefore not uncommon for the logic design to be quite costly in terms of engineering time.

In view of the above, there has been significant recent activity in the field of automatic logic synthesis. Early work centered on developing algorithms for translating a boolean function into a minimum 2-level network of boolean primitives, and extensions were developed for handling limited circuit fan-in and alternative cost functions. However, because these algorithms employ 2-level minimization, the time required to implement these algorithms increases exponentially with the number of circuits. The use of such algorithms therefore becomes impractical in designing large processors.

Other efforts have attempted to raise the level of specification, e.g., by beginning with behavioral specifications and producing technology-independent implementations at the level of boolean equations. However, the results of such techniques were usually more expensive than manual implementations and did not take advantage of the target technology. For example, the

system described by T.D. Friedman et al, in "METHODS USED IN AN AUTOMATIC LOGIC DESIGN GENERATOR (ALERT)," IEEE Trans. on Computers, Vol. C-18, No. 7 pp. 593–614 (1969), produced an implementation for an IBM 1800 processor which required 160% more gates than the manual design for that processor. Several attempts have been made to produce more efficient logic and to give the designer more control over the implementation, e.g., as described by: H. Schorr, "Toward the Automatic Analysis and Synthesis of Digital Systems," Ph.D. Thesis, Princeton University, Princeton, NJ, 1962; C.K. Mestenyi, "Computer Design Language Simulation and Boolean Translation," *Technical Report* 68–72, Computer Science Department, University of Maryland, College Park, MD, 1968; F.J. Hill and G. R. Peterson, *Digital Systems: Hardware Organization and Control*, John Wiley & Sons, Inc., New York, 1973. However, this control has resulted in specification language constraints, so that the specification is at a fairly low level and in closer correspondence with the implementation. This necessarily decreases the advantage of an automated approach, bringing it closer to a system for logic entry rather than logic synthesis.

Several tools have been developed to support the early part of the design cycle, e.g., as described in: M. Barbacci, "Automated Exploration of the Design Space for Register Transfer Systems," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1973; D. E. Thomas, "The Design and Analysis of an Automated Design Style Selector," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1977; E. A. Snow, "Automation of Module Set Independent Register-Transfer Level Design," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1978; L. J. Hafer and A. C. Parker, "Register-Transfer Level Digital Design Automation: The Allocation Process," *Proceedings of the Fifteenth Design Automation Conference*, Las Vegas, NV, 1978, pp. 213–219; A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, and J. Kim, "The CMU Desiqn Automation System - An Example of Automated Data Path Design," *Proceedings of the Sixteenth Design Automation Conference*, Las Vegas, NV. 1978, pp. 73–80. The technique described in the last-cited publication began with a functional description of a machine and produced an implementation in two technologies of the registers, register operators and their interconnections, but not the control logic to sequence the register transfers. For both TTL and CMOS implementations, however, the automated implementation required substantially more chip area than existing manual designs.

There has also been recent work in logic remapping, i.e., transforming existing implementations from one technology to another. S. Nakamura et al S. Nakamura, S. Murai, C. Tanaka, M. Terai, H. Fujiwara, and K. Kinoshita, "LORES-Logic Reorganization System," *Proceedings of the Fifteenth Design Automation Conference*, Las Vegas, NV, 1978, pp. 250–260; describe a system which will help a designer translate an existing small-or medium-scale integration implementation into large-scale integration. However, remapping usually involves one-to-one substitution of new technology primitives for old technology primitives, and this often fails to take advantage of simplification which may be available at a higher technology-independent level.

SUMMARY OF THE INVENTION

It is therefore an object of the present invention to provide an automated logic synthesis technique which overcomes the above-described drawbacks. It is a more particular object of the present invention to provide such an automated logic synthesis technique which is capable of operating at a relatively high speed while achieving end results comparable to those obtained by manual design. It is a still further object of this invention to provide such an automatic logic synthesis technique capable of achieving satisfactory results in a number of different technologies.

Briefly, these and other objects of the invention are achieved by a logic synthesis method in which a register-transfer level flowchart specification is translated in a straightforward manner into a simple AND/OR logic implementation. After expanding the logic implementation to elementary representation and then applying textbook simplifications, the simplified AND/OR implementation is translated to a NAND or NOR implementation, depending on the target technology. The NAND or NOR implementation is then simplified by applying a sequence of simplification transformations which have been found by the present inventors to achieve satisfactory results, with the transformation sequence being modified to achieve "normal," "fast" or "small" logic designs. After simplification at the NAND/NOR level, the logic implementation is then translated to the target technology and further simplified. The result is an interconnection of the primitives of the target technology in a language from which automated logic diagrams can be produced in a known manner, and which can be submitted to existing programs for automated placement and wiring and chip fabrication.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will be more clearly understood from the following description in conjunction with the accompanying drawings, wherein:

FIG. 1 is a conceptual diagram of the logic synthesis technique according to the present invention;

FIG. 2 is a chart illustrating the multiple levels of simplification in the logic synthesis technique according to the present invention;

FIGS. 3(*a*)–3(*p*) illustrate simplifying transformations at the NAND/NOR level;

FIG. 4 is a simple illustration of a portion of a flowchart specification from which the present invention begins;

FIGS. 5(*a*)–(*b*) illustrate simplifications which may be performed at the AND/OR level;

FIG. 6 is a diagram illustrating the different scenarios of simplification at the NAND/NOR level;

FIG. 7(*a*)–7(*c*) illustrate examples of simplification at the hardware level; and

FIGS. 8(*a*)–8(*b*), 9(*a*)–(*c*) and 10(*a*)–10(*c*) illustrate further examples of technology-specific hardware simplifications.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

The logic synthesis method according to the present invention is generally illustrated in FIG. 1. Previous publications describing some aspects of the system according to this invention, all of which are incorporated herein by reference, are: J. A Darringer and W. H.

Joyner, "A New Look at Logic Synthesis," *Proceedings of the Seventeeth Design Automation Conference* Minneapolis, MN, 1980, pp. 543–549; J.A. Darringer, W. H. Joyner, L. Berman, and L. Trevillyan, "Experiments in Logic Synthesis," *Proceedings of the IEEE International Conference on Circuits and Computers ICCC80*, Port Chester, NY, 1980, pp. 234–237A; J. A. Darringer, W. H. Joyner, C. L. Berman and L. Trevillyan, "Logic Synthesis Through Local Transformations," *IBM Journal of Research and Development, Vol.* 25 No. 4, July 1981.

The present invention is an automatic replacement for part of the manual design process. It operates on a logic design at three levels of abstraction. It begins with an initial implementation generated in a straightforward manner from the specification. The implementation can be simplified at this level, and then moved to the next level. This simplification is accomplished by transformations, either locally or globally, to achieve the simplification or refinement. By being able to operate on the implementation at several levels, the system can often make a small change at one level that will cause a larger simplification at a lower level. By using function-preserving transformations, it is ensured that in all cases the implementation produced will be functionally equivalent to the specified behavior. The inputs to the system illustrated in FIG. 1 are a description, in a register-transfer level, flow chart-control language, of logic functions to be implemented on a chip in a specified master slice technology, together with the interface constraints and a technology file which characterizes the target technology. The output of the system is a detailed interconnection of the primitives of the target technology in a language from which automated logic diagrams (ALD's) may be produced and which can be submitted to existing programs for automated placement and wiring and chip fabrication. The output implementation is in terms of the target technology and satisfies technology-specific constraints. Some timing or other physical problems may not be detectable before placement and wiring, and in such cases the synthesis process is repeated with a revised specification or modified constraints until an acceptable implementation is achieved.

The method according to this invention comprises PL/I programs operating on a representation of the logic in a data management system. The data management system is preferably that described by F. E. Allen et al, "THE EXPERIMENTAL COMPILING SYSTEM," IBM Journal of Research and Development, Volume 24 (1980), pages 695–715. The logic synthesis data base uses a single organization component referred to as a "box," with each box having input and output terminals which are connected by wires to other boxes. Each box also is designated by a type, which may be a primitive or may reference a definition in terms of other boxes. Thus, a hierarchy of boxes can be used, and an instance of a high-level box such as a parity box can be treated as a single box or expanded into its next-level implementation when that is desirable.

The logic synthesis data base is made of two groups of tables. The first group describes the technology being used, and is created from a technology file containing, for each box type, information such as name, function and number and names of input and output pins. These data are created in batch mode and read during initialization of the interactive system.

The second group of tables contains the representation of the logic created by the system. This group consists of a box table, a signal table and a set of auxiliary tables which describe the relationship between the boxes and the signals. There is some intentional redundancy in the data, i.e., each box has a complete list of input and output signals and each signal has a source and a list of sinks. Every box table entry contains type information which provides a link to the technology group, thus allowing programs to obtain technology information about a specific box.

Using the system generally illustrated in FIG. 1, a synthesis process according to the present invention may follow the sequence of steps shown in FIG. 2. FIG. 2 illustrates the three essential levels of description used in the method of the present invention: the initial AND/OR level **104**, a NAND or NOR level **106** (depending on the target technology), and a hardware level **108** in which the types of the boxes are books or primitives of the target technology. At every level, the implementation is a network of boxes connected by signals. The purpose of this type of implementation is to find a set of transformations and a sequence of applying these transformations such that the original functional specification could be transformed by a sequence of small steps into an acceptable implementation.

As pointed out above, the process of this invention begins at step **100** with a register-transfer level description e.g. of the type shown in FIG. 4. The description consists of two parts: a specification of the inputs, outputs and latches of the chip to be synthesized; and a flowchart-like specification of control, describing for a single clock cycle of the machine how the chip outputs and latches are set according to the values of the chip inputs and previous values of the latches. At step **102** in FIG. 2, the register-transfer level description undergoes a simple translation to an initial implementation of AND/OR logic. This AND/OR level is produced by merely replacing specification language constructs with their equivalent AND/OR implementations in a well known manner, e.g., as described in J. R. Duley, "DDL - A Digital Design Language," Ph.D. Thesis, University of Wisconsin, Madison, WI, 1968; or J. A. Darringer, "The Description, Simulations and Automatic Implementation of Digital Computer Processors," Ph. D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1969. At this first level **104** in FIG. 2, the logic begins in the form of an interconnection of boxes designated by types representing the operations which the perform, e.g., AND, OR, NOT, PARITY, EQ, XOR, DECODE, REGISTER (generic latch), SENDER, RCVR. At step **104** in FIG. 2, the initial AND/OR implementation is first expanded by taking all operators more complex than AND, OR or NOT and replacing these more complex operators with combinations of AND, OR, and NOT. Beginning with this expanded AND/OR logic, simplification is achieved by invoking PL/I program transformations which search for patterns of interconnected primitives and replace them by functionally equivalent patterns which are simpler in that they use fewer instances of operators, fewer connections, etc. The transformations at the AND/OR level **104** are local, textbook simplifications of boolean expressions, most of these simplifications reducing the number of boxes but not producing a normal form. Examples of simplifications are shown in FIGS. 5(*a*) and 5(*b*). Some of these transformations are similar to optimizing compiler techniques, e.g., constant propaga-

tion (moving "0" or "1" signals through logic blocks), common term elimination (combining blocks which compute the same function), combining nested associativecommutative operators, eliminating single input AND's and OR's, etc. Further examples of transformations used are as follows:

NOT(NOT(a))→a
AND(a, NOT(a))→0
OR(a, NOT(a))→1
OR(a, AND(NOT(a),b))→OR(a,b)
XOR(PARITY(a_1, . . . , a_n), b)→
 PARITY(a_1, . . . , a_n, b)
AND (a, 1)→a
OR(a, 1)→1

These transformations may leave fragments of logic disconnected, and this can be cleaned up in a manner similar to the way in which compilers perform dead-code elimination.

After simplification at the AND/OR level **104**, the simplified AND/OR implementation is transformed into a NAND or NOR implementation. Whereas AND/OR logic requires the use of multiple different operators in a logic design, NAND or NOR logic requires fewer operators, i.e., in a NAND logic design all logical functions can be implemented using some combination of only NAND gates. Whether a NAND or NOR implementation is produced is dependent upon the primitives available in the target technology. However, the NAND or NOR description is not technology-specific, in that there are no fan-in or fan-out restrictions. (Fan-in refers to the number of signals coming into a box, and fan-out refers to the number of sinks or destinations of a signal.) The transition to these primitives is accomplished naively by local transformations and may introduce unnecessary double NANDs or NORs which will be eliminated later. Also at this point, the chip interface information is used to place generic, i.e., not technology-specific, senders and receivers on the chip inputs and primary outputs, and to insert inverters where necessary to ensure the correct signal polarities. Techniques for accomplishing this transformation are well-known and need not be described here in detail.

At step **106** in FIG. 2, simplifying transformations are applied to each signal in the network. The NAND and NOR transforms are more difficult, and extensive experiments by the present inventors at the NAND/NOR level have resulted in a sequence or "scenario" of transformations which will produce acceptable results. The transformations are local in that they replace a small subgraph of the network (usually five or fewer boxes) with another subgraph which is functionally equivalent but simpler according to some measure. These transformations attempt to reduce the number of boxes of the implementation without increasing the number of connections. To accomplish this, the transformations must check the fan-out of the various signals involved, since this will affect the number of boxes and signals actually removed. Some of the transformations attempt to remove reconvergent fan-out which contributes to untestable stuck faults.

Some of the transformations are applied throughout the network in a number of iterations, preferably until no more transformations apply. FIGS. 3(*a*)-3(*n*) illustrate the NAND transformations NTR1 thru NTR10 used in one embodiment of this invention, and the NOR transformations would be identical except for the operator. Each transformation has an associated condition that determines if the replacement will simplify the implementation by reducing boxes or connections. These conditions depend on the fan-out of the intermediate signals and on whether the target technology is assumed to have dual-rail output.

Experiments with the NAND/NOR level transformations have resulted in a normal sequence or "scenario," of transformations which have produced acceptable results. A "fast" scenario was developed which resulted in shorter path lengths, and a "small" scenario was also developed to obtain smaller designs. These are generally indicated in FIG. 6. In the preferred embodiment of this invention, the sequence of steps in the normal NAND/NOR scenario would be as follows:

```
APPLY GENNOR: (or APPLY GENNAND);
UNTIL NOCHANGE APPLY NTR1, NTR2,
  CLEANUP, NTR3, NTR4, NTR10, CLEANUP,
  NTR7, NTR9, PROPCON, CLEANUP, CTE,
  CLEANUP;
FANIN 4;
APPLY NTR6A, FACTORN, NTR6A, CLEANUP;
APPLY NTR10, CLEANUP, NTR7 (NOIN-
  CREASE), NTR9, PROPCON, CLEANUP:
APPLY CTE, CLEANUP; FANIN 8:
APPLY NFANIN, NTR11, CLEANUP;
```

The GENNOR or GENNAND transformations merely transform the AND/OR implementation into either NAND or NOR logic in accordance with the target technology. This type of transformation is well understood in the art and need not be described in detail here.

NTR1 in FIG. 3(*a*) removes double inverters and always applies, since it is always considered desirable to reduce the number of cells, and because this transformation does not increase connects or path lengths. This transform, and others, may in some instances increase fan out, but the fan out can be reduced, if necessary, at a later point.

NTR2 in FIG. 3(*b*) applies only if s_1 has no fan out and s_2 fans out only to primitives, i.e., either NANDs or NORs. This transform will not apply if it will result in an increase in the number of connects. For example, in the transformation illustrated in FIG. 3(*b*), gates 10 and 12 are eliminated and their corresponding input and output connections are also eliminated. However, if s_2 fans out to four NANDs, it would be necessary to apply the NTR2 transformation to each one, resulting in an increase in the number of connects.

NTR3 in FIG. 3(*c*) applies only if none of the gate outputs s_i fans out, s_r does not fan out, and no gate B_i exceeds the fan-in threshold for a single-cell book. This helps set up later dotting.

NTR4 in FIG. 3(*d*) removes redundancy locally. Redundancy is a property of a combinational logic circuit, and is present when the network contains a signal that can be set to a constant value without changing the function of that network. NTR4 also replicates logic if the output s of gate 12 fans out.

NTR6A in FIG. 3(*g*) sets up dotting and is only run if dotting is allowed in the target technology.

NTR7 eliminates some forms of redundant connections. This transform will replicate boxes, if necessary, unless the parameter NOINCREASE is specified. NTR7 actually comprises three transforms illustrated in FIGS. 3(*h*)-3(*j*), all of which are run each time NTR7 is called for in the above program.

NTR9 in FIG. 3(*i*) handles cases where a signal and its negation both go to a NOR or NAND gate. The "0"

input to gate 14 will be a "1" for the equivalent NOR transformation. This transform should be followed by PROPCON, described below.

NTR10 includes two different transforms illustrated in FIGS. 3(*m*) and 3(*n*), both of which are run each time NTR10 is called for. The NTR1O transform is run only if the outputs of gates 18 and 20 and FIG. 3(*n*) do not fan out.

NTRll in FIG. 3(*o*) makes all generic registers (considered to have the OR function) have a fan-in of 1 by preceding each register with an appropriate number of primitives.

PROPCON, CLEANUP and CTE are analogous to the compiler operations of constant propagation elimination, dead-code elimination and common sub-expression elimination, respectively. Common sub-expression elimination, or common term elimination, refers to locating boxes which produce the same logic value, eliminating one box, and sharing the output of the other box.

FANIN 4 does not in itself perform any transformation but instead sets a variable known as "FANIN" to a value of 4.

FACTORN examines only boxes exceeding the FANIN limitations specified by the variable FANIN. It then applies the transformation of FIG. 3(*p*). This transformation will not reduce all boxes to below the specified FANIN limit, but only those boxes to which it applies by finding common sinks.

NFANIN corrects the fanin to the specified limit by building fanin trees which it constructs to have the fewest boxes and then to lengthen as few paths as possible.

In a NOCHANGE loop, the transformations are repeatedly run in their specified order until no further change in the logic occurs. In general, the order of the transformations and their inclusion in the NOCHANGE loop is such that suceeding transformations are invoked when preceding transformations can cause them to apply. For example, in the first loop, the sequence beginning with NTR9 is used to remove gates having complementary inputs. Since this can produce constant zeros or ones, constant propagation (PROPCON), removal of unconnected boxes (CLEANUP), common term elimination (CTE), and then more CLEANUP (to deal with now-unconnected common terms) must be run. On the other hand, after fan-in correction by factoring and NFANIN, some transformations should not be run, because they may destroy the fan-in limits already enforced.

In looking again at the program above, it can be seen that certain sequences of functions are performed, with some functions comprising a plurality of transformations. More particularly, with regard to the first NOCHANGE loop, transformations NTR1, NTR2, CLEANUP, NTR3 operate to reduce logic depth, i.e., number of levels of logic from input to output, with NTR1 reducing logic depth from two levels to one and NTR2 reducing logic depth from three levels to one. NTR3 at first glance appears to provide no depth reduction, since it transforms three levels of logic to three levels of logic. However, in some instances the last level, gate 11, can be subsequently eliminated, so that NTR3 is often useful in reducing logic depth.

Reducing logic depth, i.e., .compressing the logic into fewer levels, will increase the chance of detecting redundancy. Thus, NTR4, NTR10, CLEANUP, NTR7, NTR9, PROPCON, CLEANUP applied to remove redundancy.

After removing redundancy, a common terms elimination sequence CTE, CLEANUP is run.

After the NOCHANGE loop has finished running, transformations are applied to introduce dot patterns and to reduce fan-in to a specific level. This is accomplished by the step FANIN 4 which sets the fan-in limit to a value of 4, followed by the sequence NTR6A, FACTORN, NTR6A, CLEANUP, which serves to reduce fan-in at the expense of logic depth.

Once again, the introduction of dot patterns and the factoring to reduce fan-in may result in redundancy, so that the redundancy removal sequence NTR10, CLEANUP, NTR7, NTR9, PROPCON, CLEANUP is applied.

Common terms are then eliminated by running CTE, CLEANUP.

Finally, the logic must be adjusted to the maximum fan-in value permitted by the target technology, e.g., a fan-in value of 8. This is achieved by applying FANIN 8 to set the fanin value at 8 followed by NFANIN, CLEANUP.

As should now be appreciated, the above program can be functionally represented as follows:

A. LOGIC DEPTH REDUCTION LOOP
   A1. REDUCE LOGIC DEPTH
   A2. REMOVE REDUNDANCY
   A3. ELIMINATE COMMON TERMS
B. INTRODUCE DOT PATTERNS AND FACTOR TO REDUCE FANIN TO SPECIFIC LEVEL
C. REMOVE REDUNDANCY
D. ELIMINATE COMMON TERMS
E. ADJUST LOGIC TO MAXIMUM PERMITTED FANIN

The operations subsequent to the logic depth reduction loop may tend to expand the logic depth, so that the above process can generally be seen as a compression stage followed by an expansion stage. While it may be theoretically possible to obtain maximum logic depth reduction through two-level boolean minimization, this would compress the logic so far that re-expansion to take advantage of other simplifying transformations, e.g., at the subsequent hardware simplification, would be much more difficult. Thus, the logic compression transforms have been found particularly suitable.

The program set forth above concerns a normal scenario, and the "fast" and "small" scenarios can be obtained by modifying the above program as follows: for the small scenario, the following additional NOCHANGE loop is inserted after the NOCHANGE loop in the normal scenario:

UNTIL NOCHANGE APPLY NTR6, NTR5, NTR1, NTR2, CLEANUP, NTR3, NTR4, NTR10, CLEANUP, NTR7, NTR9, PROPCON CLEANUP, CTE, CLEANUP;

NTR5 in FIG. 3(*e*) applies only if the number of cells does not increase, and NTR6 in FIG. 3(*f*) applies only if the number of cells is decreased. Inspection of NTR5 and NTR6 shows that they can increase path length, and they are consequently only used in the small scenario. The other transformations in the added loop are provided to act on any changes which may result from NTR5 and NTR6. For example, NTR5 and NTR6 can produce double inverters, so the sequence beginning with NTR1 is run. NTR1 eliminates double inverters, and can introduce situations where other transforms apply.

Examination of the second NOCHANGE loop set forth above will reveal that the loop includes a first sequence NTR6, NTR5 for reducing the cell count by increasing the logic depth. The sequence NTR1, NTR2, CLEANUP, NTR3 is then applied to mitigate the logic depth reduction by taking advantage of transforms made available by NTR6, NTR5. After this logic depth reduction sequence, the redundancy removal and common term elimination sequence are applied in the first NOCHANGE loop.

Thus, the program for the "small" scenario can be written:

A. LOGIC DEPTH REDUCTION LOOP
   A1. REDUCE LOGIC DEPTH
   A2. REMOVE REDUNDANCY
   A3. ELIMINATE COMMON TERMS
   A'. CELL COUNT REDUCTION LOOP
   A1'. REDUCE CELL COUNT
   A2'. REDUCE LOGIC DEPTH
   A3'. REMOVE REDUNDANCY
   A4'. ELIMINATE COMMON TERMS
B. INTRODUCE DOT PATTERNS AND FACTOR TO REDUCE FANIN TO SPECIFIC LEVEL
C. REMOVE REDUNDANCY
D. ELIMINATE COMMON TERMS
E. ADJUST LOGIC TO MAXIMUM PERMITTED FANIN

While the "small" scenario is designed to emphasize minimization of gate count, the "fast" scenario is designed to emphasize shorter path lengths, sometimes at the expense of gate count. Path length refers to the delay along a path from a signal's source to one of its destinations. Usually, path lengths are measured from registers or primary chip inputs to registers or primary chip outputs. The result can be the number of boxes in the path or the estimated delay of that path in nanoseconds.

The fast scenario inserts a call to NTR8 as the last step run in the first NOCHANGE loop. Immediately thereafter, FANIN is set to a value of 8 rather than 4, and NTRll is omitted from the last line of the program. The significance of these changes to the program is as follows:

NTR8 in FIG. 3(*k*) is used in the fast scenario because it shortens paths. This may sometimes be at the expense of cells, however, since some of the boxes shown in FIG. 3(*k*) may have to be replicated. The factoring to a fanin of 8, also produces shorter paths, but may increase the cell count, e.g., in a dual-rail technology in which a 4-way NOR/OR required one cell and an 8-way required two cells. This will be explained in more detail with reference to FIGS. 8(*a*) and 8(*b*).

In a particular technology, there may be a number of different primitives or "books" having different fan-in capabilities, and different books may include different numbers of cells. For example, an 8-way NAND gate may use two cells while a 4-way NAND gate may use one cell. If 8-way NAND gates are used, e.g., to combine ten different inputs in two combinations with four inputs common to each combination, the result may be as shown in FIG. 8(*a*). Each book would receive seven inputs, and a total of four cells would be used.

If fan-in is limited to a value of 4, the same logic could be implemented as shown in FIG. 8(*b*) using three 4-way books. Although the number of books has increased, each book includes only one cell, so that the cell count decreases from four to three. However, the cell count decrease is at the expense of increasing the logic depth by one level.

In the "normal" or "small" scenarios, it is worthwhile to set the fan-in value to 4 and to factor in an attempt to take advantage of the cell reduction which may be realized by using the smaller books. In the "fast" scenario, however, the increase in logic depth accompanying the use of the smaller books is unacceptable, and the fanin is instead set to the maximum allowable fan-in after the NOCHANGE loop.

Thus, the simplification program for the "fast" scenario can be functionally described as follows:

```
LOGIC DEPTH REDUCTION LOOP
  A1. REDUCE LOGIC DEPTH
  A2. REMOVE REDUNDANCY
  A3. ELIMINATE COMMON TERMS
  A4. REDUCE LOGIC DEPTH WHILE INCREASING CELL COUNT
B. INTRODUCE DOT PATTERNS AND FACTOR TO REDUCE FANIN TO MAXIMUM ALLOWED BY TECHNOLOGY
C. REMOVE REDUNDANCY
D. ELIMINATE COMMON TERMS
E. ADJUST LEVEL TO MAXIMUM PERMITTED FANIN
```

The search strategy for the above transformations is to search the interconnected boxes of the date-base in sequence, looking for a pattern to which the transform may apply. The search is done for each transform in an efficient way, e.g., NTR2 searches the entire logic design for a one-input inverter, since this is faster than examining each multi-way NAND or NOR to determine if an inverter precedes or follows it.

After the simplification sequence described above, transformations are applied to the logic to map the NAND or NOR implementation to the target technology, simplify the technology-specific implementation, and enforoe technology-specific restrictions. This is performed at level 108 in FIG. 2. The transformations applied at level 108 may be generally described as follows, although their exact implementation will depend on the target technology

Technology-specific transforms may preferably be applied in the following order: first, generic NAND/-NOR gates are mapped to their counterparts in the target technology. If the fan-in of a gate is too high and there is no corresponding primitive in the technology, a tree of primitives must be built to produce the same logical function. REG's, the generic latches, are mapped to the technology-specific latches. In general, the technology-specific latches have a limited number of pins for data values. If more data values are gated into the latch than can be accomodated, extra "ports" must be connected to the latch in a manner prescribed for the technology. SENDER's and RECEIVER's are mapped to their technology-specific counter parts.

Second, if the target technology is dual-rail, dual-rail books are introduced. With both positive and negative phases available from each gate, all inverters (except those on chip inputs) are removed and their output signals connected to the opposite phase of the source of their input signals. Third, technology-specific "tricks" are introduced, e.g., special books, drivers, receivers, etc., which were not known at the time of the generic transformation. These implement certain functions, such as XOR, combinations of driver and logic functions, combinations of receiver and logic funtions, combinations of latch and receiver, etc., using fewer cells

than the primitive NAND or NOR implementation. The pattern of technology-specific NAND's or NOR's is searched for and replaced by the appropriate block. In FIG. 7(*a*), three cells can be replaced with a single NAND in the target technology having a built-in receiver.

If dots, i.e., wired AND's or OR's, are allowed in the target technology, patterns implementing +AND or +OR functions are located. If the inputs of these patterns have no fan-out, the pattern is replaced by a dot, e.g., as shown in FIG. 7(*b*). Dots can also be introduced to reduce fan-in as shown in FIG. 7(*c*). After dots are introduced, more special books may be present and are searched for again.

Next, fan-out is adjusted to meet constraints. Fanout limits are specified for technology-specific box types and output pins of those boxes. Fan-out is brought within these limits by replicating the violating box and distributing some of its fan-out to the copy (parallel repowering) or driving some of the fan-out with a repowering +OR or +AND function in front of the violating box (serial repowering). Additional dual-rail books are added after fan-out adjustment, but not so as to violate fan-out constraints.

Next, clock signals are introduced as chip inputs and distributed to the latches of the chip according to technology-specific requirements for clock distribution. Depending on the technology, this requires clock balancing and introduction of special clock drivers.

Next, path lengths back from latch-to-latch and from chip output to chip input are analyzed. Long paths are first shortened by rearranging fan-in and fan-out repowering, introducing dots (even at the cost of cell count), "undoing" factoring transformations performed at a higher level and introducing high-power books. Short paths are then padded to meet minimum path length requirements.

The fan-out adjustment is then repeated to correct fan-out violations which may have resulted from the path length correction.

Finally, scan-in and scan-out pins are introduced and the latches are linked together in an LSSD scan ring. A chip-in-place test and/or inhibit signals are introduced for chip outputs where required. Since this may introduce fan-out violations, fan-out adjustment is repeated.

An example of a hardware conversion and simplification program for a NOR technology following the above-described sequence may be as follows:

```
APPLY GENHW, CLEANUP;
APPLY DUAL (NO LIMIT), CLEANUP;
APPLY OPTDRIVE (SMALL OR FAST),
  CLEANUP,
OPTXOR (SMALL OR FAST), CLEANUP;
APPLY GENDOT;
APPLY OPTDRIVE (SMALL OR FAST),
  CLEANUP, OPTXOR (SMALL OR FAST),
  CLEANUP;
APPLY FANOUT, DUAL, CLEANUP;
APPLY CLOCK;
APPLY TIMINE;
APPLY FANOUT, DUAL, CLEANUP
APPLY SCANP, FANOUT;
```

GENHW maps generic gates to hardware primitives. Since fan-in has been adjusted at the end of the NAND/NOR simplification, much of this step is merely one-to-one mapping.

DUAL removes necessary inverters in dual-rail technologies by absorbing the inverters into other gates which already have positive and negative phases available. This transform will normally be applied so as to exceed the fan-out limit. However, with the NOLIMIT option this transform will always apply.

OPTDRIVE takes advantage of a technology-specific book available, i.e., a driver book with built-in NOR capability. As shown in FIG. 9(*a*), the logic design may at this point include a NOR gate with a branched output with one branch going to a driver. Since both functions can be served by a single book in the target technology, the arrangement of FIG. 9(*b*) can be substituted. However, while this may be desirable in "normal" and "small" scenarios, there is a sacrifice in speed. Thus, for a "fast" scenario, the transformation is to the arrangement shown in FIG. 9(*c*) which provides for "parallel" operation and therefore higher speed at the expense of cell count.

OPTXOR takes advantage of a further technologyspecific book, i.e., the XOR book. This transformation searches for a pattern of NOR gates providing the XOR function, e.g., as shown in FIG. 10(*a*), and substitutes the XOR book as shown in FIG. 10(*b*). Again, however, the transformation to FIG. 10(*c*) is employed in the "fast" scenario.

GENDOT introduces dotting in such a manner as to both eliminate gates and reduce fan-in. E.g., the transformation shown in FIG. 7(*b*) will eliminate gates **15** and **16** while the transformation shown in FIG. 7(*c*) will not eliminate gate **17** but will reduce the fan-in to that gate. This may save cells by permitting the use of a smaller book in the target technology and by allowing other transforms to apply. Since GENDOT changes the logic, OPTDRIVE and OPTXOR are applied again to search for more special books which may now exist.

FANOUT is applied to reduce the fan-out to the allowed limit. Note that the first half of the above hardware level simplification program is run without regard to fan-out limitations, as even the DUAL transform is applied with its NOLIMIT option. The various transformations may have caused fan-out violations which should be corrected by applying FANOUT in the manner discussed above. DUAL is then applied again, but this time so as not to violate fan-out constraints.

CLOCK is applied to distribute clock signals according to technology specific requirements in a manner known in the art.

TIMING is applied to correct path lengths by rearranging fan-in and fan-out trees, introducing more dots and chaning power levels to shorten long path lengths, and inserting pad logic to lengthen the short paths, as necessary. After TIMING is applied, fan-out adjustment is again performed to correct any violations which may have resulted from the timing correction, and DUAL is again run, within fan-out constraints, to take advantage of changes made during fan-out adjustment.

Finally, SCANP is applied to link the registers in a LSSD scan path. Fan-out is again adjusted to correct any violations which may have resulted from SCANP.

The logic synthesis system of this invention employs three different levels of simplification between the original specification and the final implementation: high level simplifications, NAND/NOR level simplifications and technology specific simplifications. Several of the transforms at the three different levels are analogous, differing only in the types of boxes to which they apply, so that simplifications not made at one level would be caught later. This may appear redundant, but the application of transforms as early as possible reduces the size

of the implementation and helps prevent a greater explosion in size when, e.g., conversion to NANDs takes place.

A significant advantage of the present invention resides in its adaptibility to more than one technology, requiring modifications to only a part of the system and leaving the technology-independent portions intact. This makes the synthesis process according to the present invention useful in synthesizing logic in a number of different technologies, and in fact facilitates the remapping from one technology to another in an efficient manner. Rather than merely mapping hardware primitives one-to-one from one technology to another, a first technology implementation is abstracted to a technology-independent level, e.g., from a TTL chip implementation to a NAND level implementation with generic registers, drivers and receivers. The NAND implementation can be mapped to a NOR level implementation in a straight-forward manner, with the NOR level simplification being performed in the manner described above with reference to level **106** in FIG. **2**. The hardware mapping and simplification can then be performed in the manner described with reference to level **108** in FIG. **2**. This enables the remapping to take advantage of simplifications which may be available at the NOR level.

Some of the work described in the earlier-cited publications concerned a synthesis process beginning with a behavioral description and producing technology-independent implementations of boolean equations. These processes did not take advantage of the target technology. Other work has centered on the synthesis of the data-flow portion of a machine, synthesis from a high-level behavioral description to a register-transfer description, and implementation of control logic in microcode or programmable logic arrays. In contrast, the present invention provides the following significant features:

First, the present invention uses local transformations at several levels of description, passing through technologyindependent levels of description to a technology-specific description. This enhances the simplification while also facilitating the re-implementation of a design in a different technology.

Second, the specific sequences of simplifying transformations and the conditions associated with them have been found to provide acceptable results in normal, fast and small scenarios, thus making automated logic synthesis practical.

Further, timing, driver and other interface constraints are used at the hardware level to generate logic meeting these requirements.

Still further, the automated logic synthesis process according to the present invention greatly facilitates timing analysis and correction of the design to remove path length problems.

What is claimed is:

1. An automated logic synthesis method of designing, on a computer, a logic circuitry implementation in a desired technology from input data to said computer comprising a description of operating characteristics to be provided by said logic circuitry, said method comprising the steps of:

generating, via said computer, a first logic circuit design in a first logic system in accordance with said description;

simplifying said first logic design via said computer;

converting, via said computer, said simplified first logic design to a second logic design in a second logic system requiring fewer different logic operators than said first logic system, said second logic system comprising a plurality of interconnected cells and performing equivalent functions;

simplifying, via said computer, said second logic design, said step of simplifying said second logic design comprising the steps of: applying a depth reduction sequence of logic transformations for reducing the depth of said second logic design; and subsequently applying a size reduction sequence of logic transformations for reducing the size while possibly increasing the depth of said second logic design; and

converting, via said computer, said simplified second logic design to said desired technology.

2. A method as defined in claim 1, wherein said step of applying said depth reduction sequence of logic transformations comprises: applying a first logic transformation set for reducing logic depth; applying a second logic transformation set for reducing redundancy; and applying a third logic transformation set for eliminating common terms.

3. A method as defined in claim 2, wherein said step of applying said size reduction sequence of logic transformations comprises: applying a fourth logic transformation set for reducing logic size while possibly increasing logic depth; applying a fifth logic transformation set for reducing redundancy; and applying a sixth logic transformation set for eliminating common terms.

4. A method as defined in claim 3, wherein said second and fifth logic transformation sets include at least one common logic transformation. said common logic transformation being applied in said depth reduction sequence regardless of whether said common logic transformation will increase the number of cells in said second logic design and being applied in said size reduction sequence only if it will not result in an increase in said number of cells.

5. A method as defined in claim 2, wherein said step of applying said depth reduction sequence of logic transformations further comprises applying a fourth logic transformation set (e.g.. NTR8), following said third logic transformation set, for further reducing said logic depth while increasing the number of cells in said second logic design.

6. A method as defined in claim 3, wherein said converting step comprises converting said simplified second logic design to a hardware logic design in said desired technology represented as a plurality of hardware primitives. said method further comprising the step of simplifying said hardware design, said hardware simplifying step comprising:

applying a first hardware transformation set for substituting technology-specific books for predetermined patterns of said hardware primitive;

dotting signal lines to decrease the number of components in said hardware logic design, and to decrease fan-in in some portions of said hardware logic design even if the number of components in said portions is not decreased;

applying said first hardware transformation set;

correcting fan-out in said hardware logic design to a desired value;

adjusting path lengths in said hardware logic design; and

correcting fan-out to said desired value.

7. A method as defined in claim 1, wherein said step of applying said size reduction sequence of logic transformations comprises applying a first logic transformation (e.g., FACTORN) for reducing a fan-in characteristic of some portions of said second logic design in accordance with a first fan-in value.

8. A method as defined in claim 1, further comprising the step of applying a cell reduction sequence of logic transformations for reducing the number of cells in said second logic design, said cell reduction sequence being applied between said depth reduction and size reduction sequences.

9. A method as defined in claim 8, wherein said cell reduction sequence of logic transformations includes a first set of logic transformations followed by a second set of logic transformations, said second set of logic transformations comprising said depth reduction sequence of logic transformations.

10. A method as defined in claim 8, wherein said desired technology has a maximum allowable fan-in value, said step of applying said size reduction sequence of logic transformations comprising applying a first logic transformation (e.g., FACTORN) for reducing a fan-in characteristic of some portions of said second logic design in accordance with a desired fan-in value less than said maximum allowable fan-in value.

11. A method as defined in claim 10, further comprising the step of correcting the fan-in characteristics of said second logic design in accordance with said maximum allowable fan-in value subsequent to application of said size reduction sequence of logic transformations.

12. A method as defined in claim 1, wherein said depth reduction sequence of logic transformations is applied a plurality of times prior to applying said size reduction sequence of logic transformations.

13. A method as defined in claim 1, wherein said first logic design is implemented in AND/OR logic and said second logic design is implemented in NAND logic.

14. A method as defined in claim 1, wherein said first logic design is implemented in AND/OR logic and said second logic design is implemented in NOR logic.

15. An automated logic synthesis method of designing, on a computer, a logic circuitry implementation in a desired technology from input data to said computer comprising a description of operating characteristics to be provided by said logic circuitry, said method comprising the steps of:

generating, via said computer, a first logic circuit design in a first logic system in accordance with said description;

simplifying said first logic design via said computer;

converting, via said computer, said first logic design to a second logic design in a second logic system requiring fewer different logic operators than in said first logic design, said second logic design comprising a plurality of interconnected cells and performing equivalent functions as said first logic design;

simplifying said second logic design via said computer;

converting, via said computer, said simplified second logic design to a hardware design in said desired technology comprising a plurality of interconnected hardware components; and

simplifying said hardware design via said computer, said step of simplifying said hardware design comprising: applying a first hardware transformation set for substituting technology specific components for predetermined patterns of said hardware;

dotting signal lines via said computer, to decrease the number of components in said hardware logic design and to decrease fan-in in some portions of said hardware logic design even if the number of components is said portions is not decreased;

adjusting path lengths in said hardware logic design via said computer; and

correcting fan-out in said hardware logic design to a desired value via said computer.

16. A method as defined in claim 15, wherein said step of simplifying said hardware design further comprises applying said first hardware transformation set again after said dotting step but before said adjusting step.

17. A method as defined in claim 16, wherein said step of simplifying said hardware logic design further comprises the step of correcting said fan-out in said hardware logic design after said second application of said first hardware transformation set and before said adjusting step

18. A method as defined in claim 15, wherein said hardware logic design includes inverters receiving and inverting outputs from associated components, and wherein, when said desired technology is a dual-rail technology, said step of simplifying said hardware design further comprises the step of applying a dual-rail transformation for removing some of said inverters by substituting for said inverter and opposite-phase output signal available from its associated component, said dual-rail conversion transformation being applied both prior to said step of applying said first hardware transformation and subsequent to said step of correcting fan-out.

19. A method as defined in claim 18, wherein said dual-rail conversion transformation is applied prior to said step of applying said first hardware transformation without regard to the effect of said dual-rail conversion transformation on fan-out characteristics of said hardware logic design, said dual-rail conversion transformation being applied after said step of correcting fan-out only to the extent that application of said dualrail conversion transformation will not result in fan-out exceeding said desired value.

20. A method as defined in claim 15, wherein said hardware logic design includes inverters receiving and inverting outputs from associated components, and wherein, when said desired technology is a dual-rail technology, said step of simplifying said hardware design further comprises the step of applying a dual-rail comversion transformation, prior to said step of applying said first hardware transformation, for removing some of said inverters by substituting for said some inverters an oppositephase output available from their associated components.

21. A method as defined in claim 20, wherein said step of simplifying said hardware design further comprises the step of applying said dual-rail conversion transformation subsequent to said step of correcting fan-out.

22. A method as defined in claim 21, wherein said dual-rail conversion transformation is applied prior to said application of said first hardware transformation without regard to the effect of said dual-rail conversion transformation on fan-out characteristics of said hardware logic design, and is applied subsequent to said step of correcting fan-out only to the extent that application of said dual-rail conversion transformation will not result in fan-out exceeding said desired value.

23. An automated logic synthesis method of designing, on a computer, a logic circuitry implementation in a desired technology from input data to said computer comprising a description of operating characteristics to be provided by said logic circuitry, said method comprising the steps of:

generating a first logic circuit design via said computer, in accordance with said description;

simplifying said first logic design via said computer;

converting, via said computer, said first logic design to a second logic design in a second logic system requiring fewer different logic operators than in said first logic design;

simplifying said second logic design via said computer;

converting, via said computer, said simplified second logic design to a hardware design in said desired technology comprising a plurality of interconnected hardware components and including inverters for receiving and inverting output signals from associated ones of said components; and

simplifying said hardware design via said computer, said step of simplifying said hardware logic design comprising: applying a dual-rail conversion transformation for removing some of said inverters by substituting for said some inverters an opposite phase output signal available from their associated components, said dual-rail conversion transformation being applied without regard to the effect on fan-out characteristics of said hardware logic design; applying a first hardware transformation set for substituting technology-specific components; dotting signal lines in said hardware logic design; adjusting path lengths in said hardware logic design; correcting fan-out in said hardware logic design to a desired value; and applying said dual-rail conversion transformation only to the extent that it does not result in a fan-out exceeding said desired value.

24. An automated logic synthesis method of designing, on a computer, a logic circuitry implementation in a desired technology from input data to said computer comprising a description of operating characteristics to be provided by said logic circuitry, said method comprising the steps of:

generating, via said computer, a first logic circuit design accordance with said description;

simplifying said first logic design via said computer;

converting, via said computer, said first logic design to a second logic design in a second logic system requiring fewer different logic operators than in said first logic design;

simplifying said second logic design via said computer;

converting, via said computer, said simplified second logic design to a hardware design in said desired technology comprising a plurality of interconnected hardware components; and

simplifying said hardware design via said computer, said step of simplifying said hardware logic design comprising selectively applying first first or second hardware transformation sets for substituting technology-specific components for predetermined patterns of said hardware components, said first hardware transformation set resulting in fewer components than said second hardware transformation set and said second hardware transformation set resulting in higher-speed logic than said first hardware transformation set.

* * * * *

United States Patent [19]

Dunn

[11] Patent Number: 4,656,603

[45] Date of Patent: Apr. 7, 1987

[54] SCHEMATIC DIAGRAM GENERATING SYSTEM USING LIBRARY OF GENERAL PURPOSE INTERACTIVELY SELECTABLE GRAPHIC PRIMITIVES TO CREATE SPECIAL APPLICATIONS ICONS

[75] Inventor: **Robert M. Dunn**, Woodbridge, Conn.

[73] Assignee: **The Cadware Group, Ltd.**, New Haven, Conn.

[21] Appl. No.: **585,535**

[22] Filed: **Mar. 1, 1984**

[51] Int. Cl.⁴ G06F 3/153; G06F 15/40

[52] U.S. Cl. 364/900; 364/488; 364/521; 340/721; 340/747

[58] Field of Search ... 364/200 MS File, 900 MS File, 364/300, 513–515, 521; 340/721, 747, 750, 799

[56] References Cited

U.S. PATENT DOCUMENTS

4,315,315	2/1982	Kossiakoff	364/300
4,455,619	6/1984	Masul et al.	364/900
4,546,435	10/1985	Herbert et al.	364/300
4,555,772	11/1985	Stephens	364/900

FOREIGN PATENT DOCUMENTS

0100798	2/1984	European Pat. Off.	364/521
2941824	4/1980	Fed. Rep. of Germany	364/300

OTHER PUBLICATIONS

D. Laidlaw, *Graphic Entity Transformation & Manipulation,* IBM Technical Disclosure Bull., (vol. 21, No. 3, Aug. 1978), pp. 1234–1243.

P. Groner, *Computer Aided Design of VLSI Saves Man-Hours, Reduces Errors,* Control Engineering (Apr. 1981), pp. 55–57.

R. A. Armstrong, *Applying CAD to Gate Arrays Speeds 32-Bit Mini Computer Design,* Electronics (Jan. 13, 1981), pp. 167–173.

D. R. Fullenwider et al, *Computer Graphics & the Practice of Architecture,* IEEE Computer Graphics (Oct. 1981) pp. 19–26.

Primary Examiner—Archie E. Williams, Jr.
Attorney, Agent, or Firm—Barry R. Lipsitz

[57] ABSTRACT

An interactive rule based system enables problem solutions to be generated in schematic diagram form. A methodology designer selects and arranges graphic primitives using a graphics terminal to create a library of icons. Under control of a computer processor, the methodology designer is prompted to identify, by way example, parameters for using each icon. The system generates and stores a specific set of rules pertaining to the use of each icon on the basis of the parameters identified. The stored rules are cross-referenced to the icon to which they pertain, so that whenever the icon is selected by a problem solving user for use in building a problem solution, the rules pertaining thereto will be accessed and applied. A methodology designer can also select and concatenate functions to each other and to icons to create more complex functions for use in building problem solutions. New functions can also be created in the form of truth tables which establish a transfer function across an icon. By accessing and selecting icons and functions created by a methodology designer, a problem solving user can build a solution to a problem by graphically coupling the icons and functions together on a chart.

19 Claims, 16 Drawing Figures

Microfiche Appendix Included
(3 Microfiche, 200 Pages)

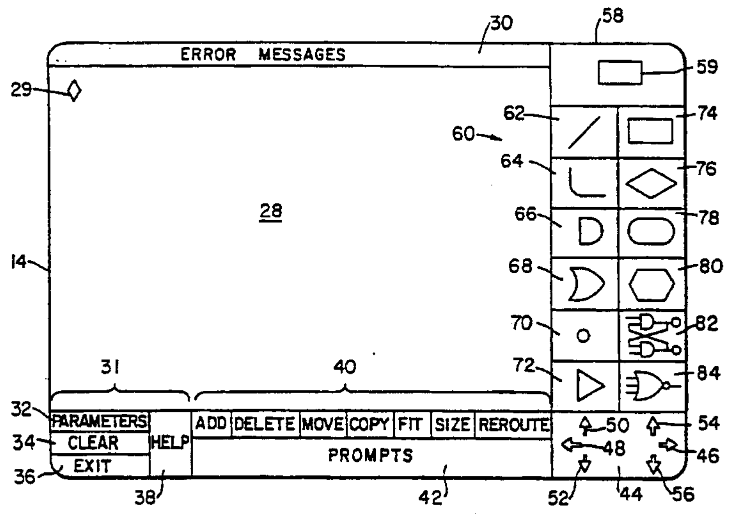
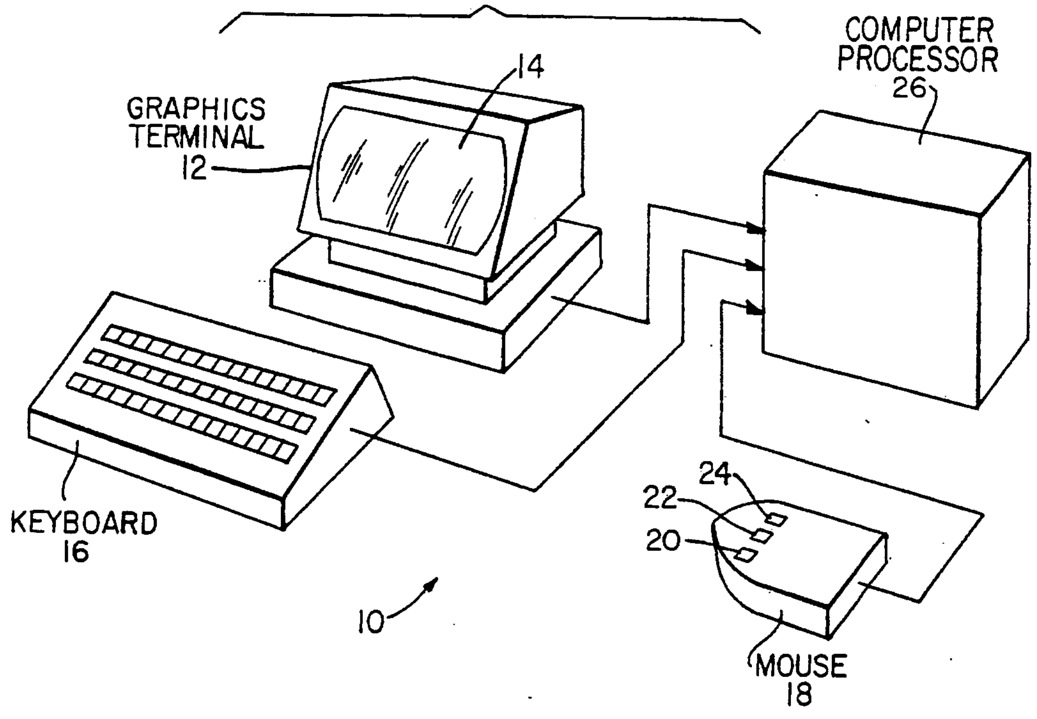
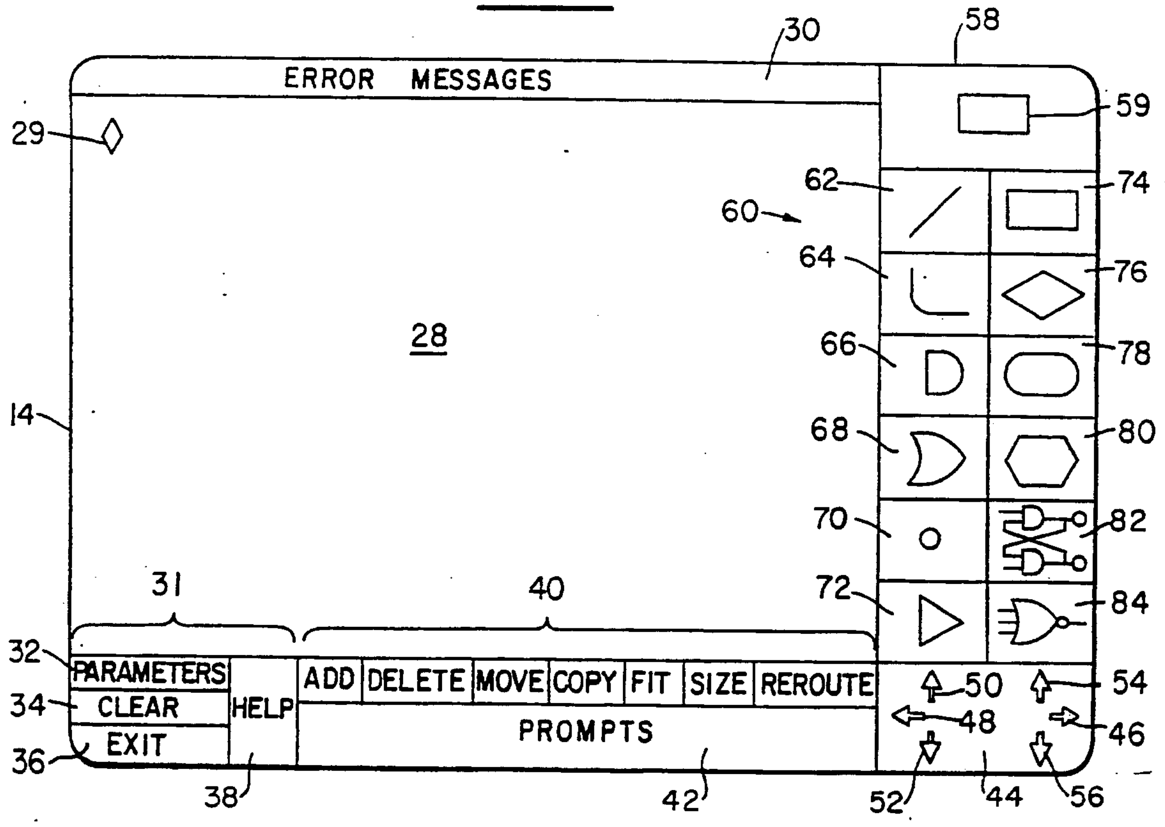


FIG. 1



GRAPHICS TERMINAL 12
14
COMPUTER PROCESSOR 26
KEYBOARD 16
24
22
20
MOUSE 18
10

FIG. 2



30
58
ERROR MESSAGES
59
29
62
74
60
64
76
28
66
78
14
68
80
70
82
72
84
31
40
32
PARAMETERS
34
CLEAR
EXIT
36
HELP
38
ADD
DELETE
MOVE
COPY
FIT
SIZE
REROUTE
PROMPTS
42
50
54
48
46
52
44
56

FIG. 3

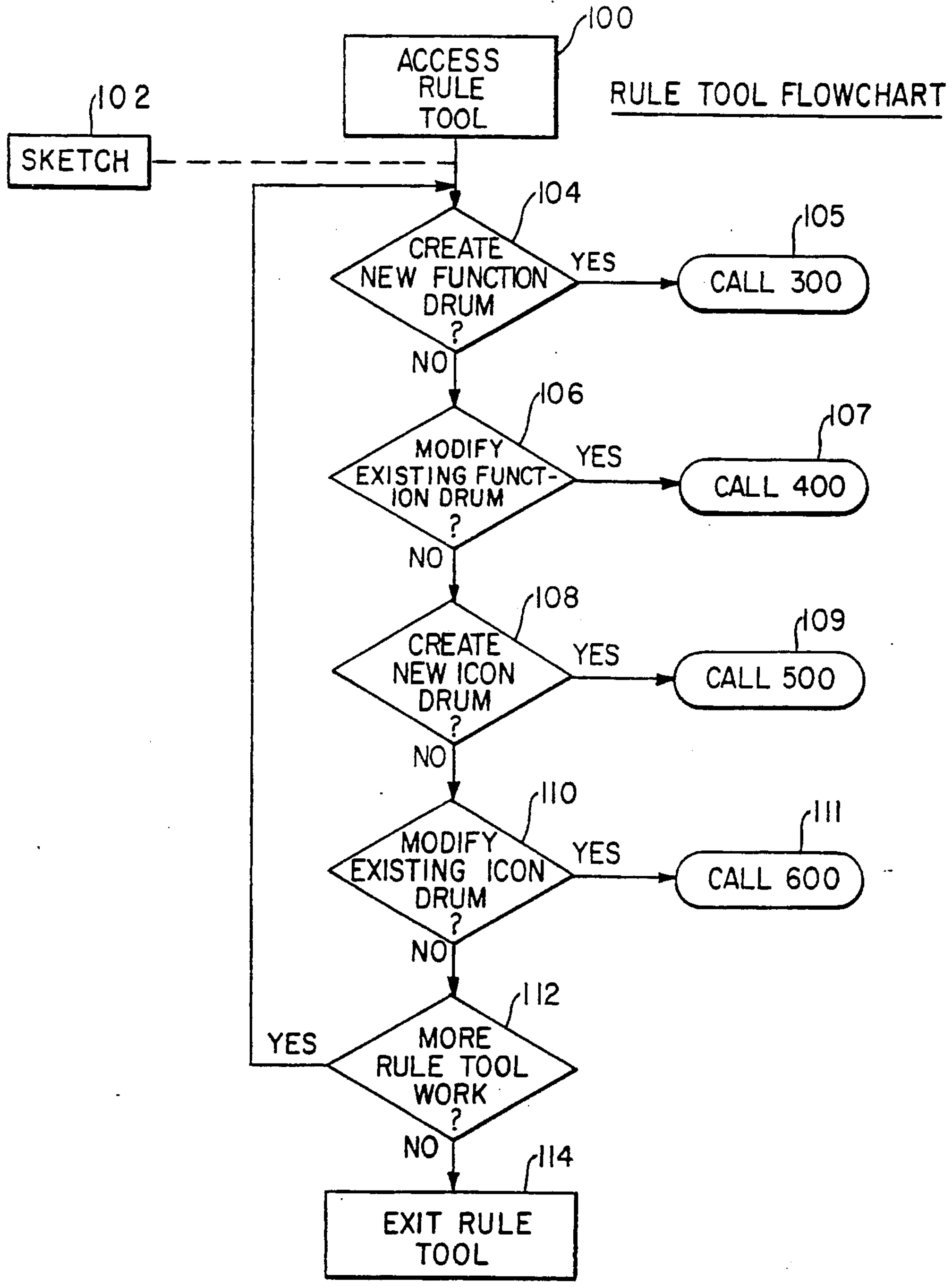
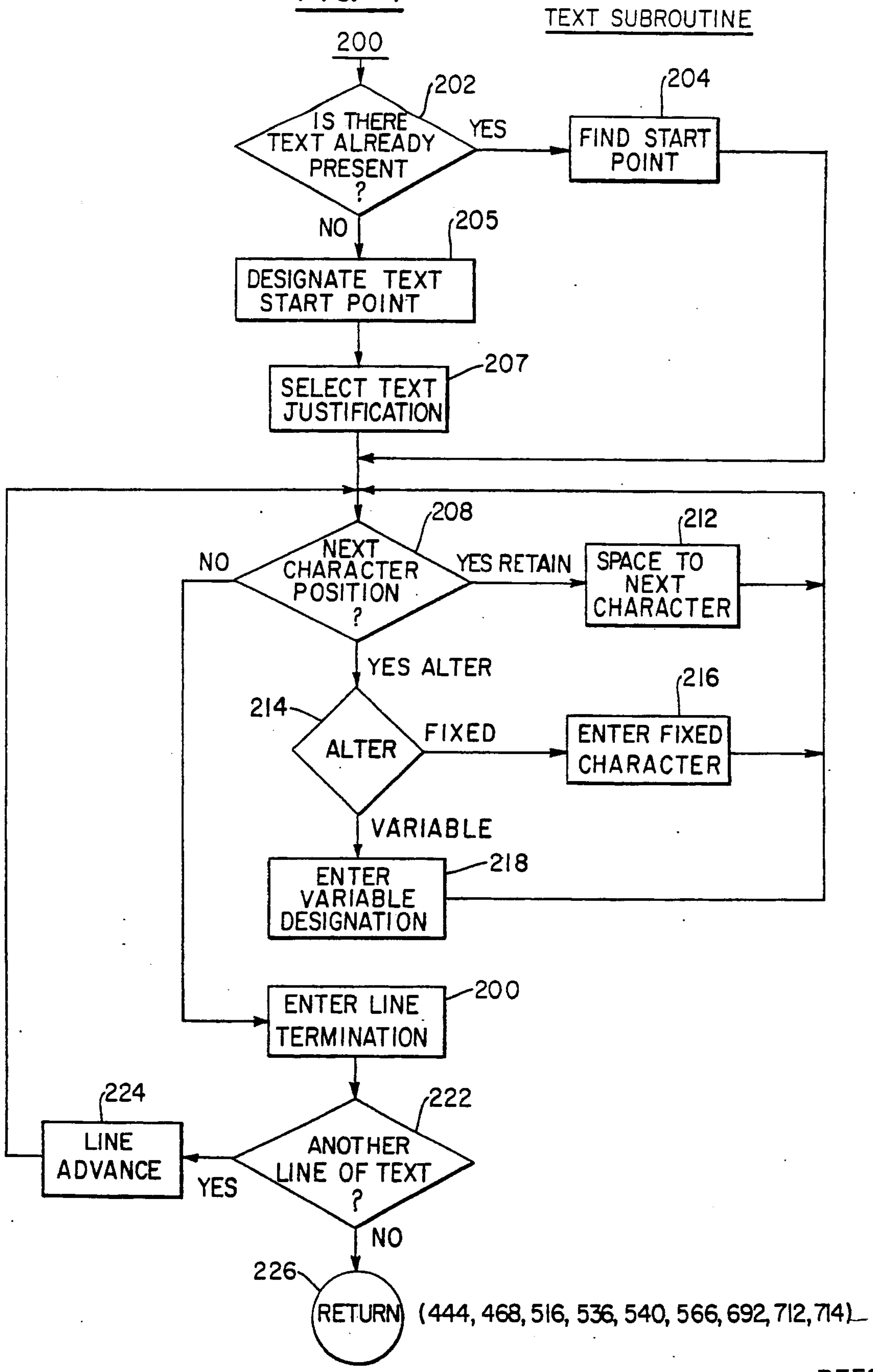

RULE TOOL FLOWCHART
100
ACCESS RULE TOOL
102
SKETCH
104
CREATE NEW FUNCTION DRUM ?
YES
105
CALL 300
NO
106
MODIFY EXISTING FUNCTION DRUM ?
YES
107
CALL 400
NO
108
CREATE NEW ICON DRUM ?
YES
109
CALL 500
NO
110
MODIFY EXISTING ICON DRUM ?
YES
111
CALL 600
NO
112
MORE RULE TOOL WORK ?
YES
NO
114
EXIT RULE TOOL

FIG. 4



TEXT SUBROUTINE
200
202
IS THERE TEXT ALREADY PRESENT ?
YES
204
FIND START POINT
NO
205
DESIGNATE TEXT START POINT
207
SELECT TEXT JUSTIFICATION
208
NEXT CHARACTER POSITION ?
NO
YES RETAIN
212
SPACE TO NEXT CHARACTER
YES ALTER
214
ALTER
FIXED
216
ENTER FIXED CHARACTER
VARIABLE
218
ENTER VARIABLE DESIGNATION
200
ENTER LINE TERMINATION
222
ANOTHER LINE OF TEXT ?
YES
224
LINE ADVANCE
NO
226
RETURN
(444, 468, 516, 536, 540, 566, 692, 712, 714)

FIG. 5

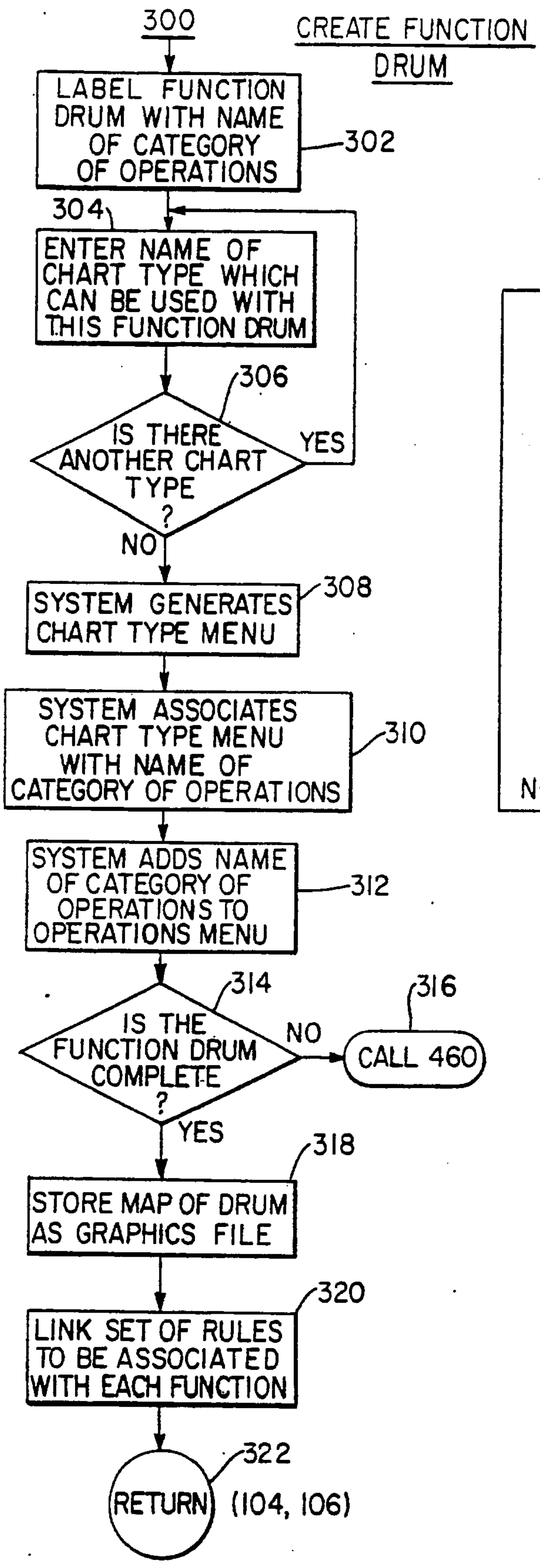

300
CREATE FUNCTION DRUM
LABEL FUNCTION DRUM WITH NAME OF CATEGORY OF OPERATIONS
302
304
ENTER NAME OF CHART TYPE WHICH CAN BE USED WITH THIS FUNCTION DRUM
306
IS THERE ANOTHER CHART TYPE ?
YES
NO
SYSTEM GENERATES CHART TYPE MENU
308
SYSTEM ASSOCIATES CHART TYPE MENU WITH NAME OF CATEGORY OF OPERATIONS
310
SYSTEM ADDS NAME OF CATEGORY OF OPERATIONS TO OPERATIONS MENU
312
314
IS THE FUNCTION DRUM COMPLETE ?
NO
316
CALL 460
YES
318
STORE MAP OF DRUM AS GRAPHICS FILE
320
LINK SET OF RULES TO BE ASSOCIATED WITH EACH FUNCTION
322
RETURN
(104, 106)

FIG. 6

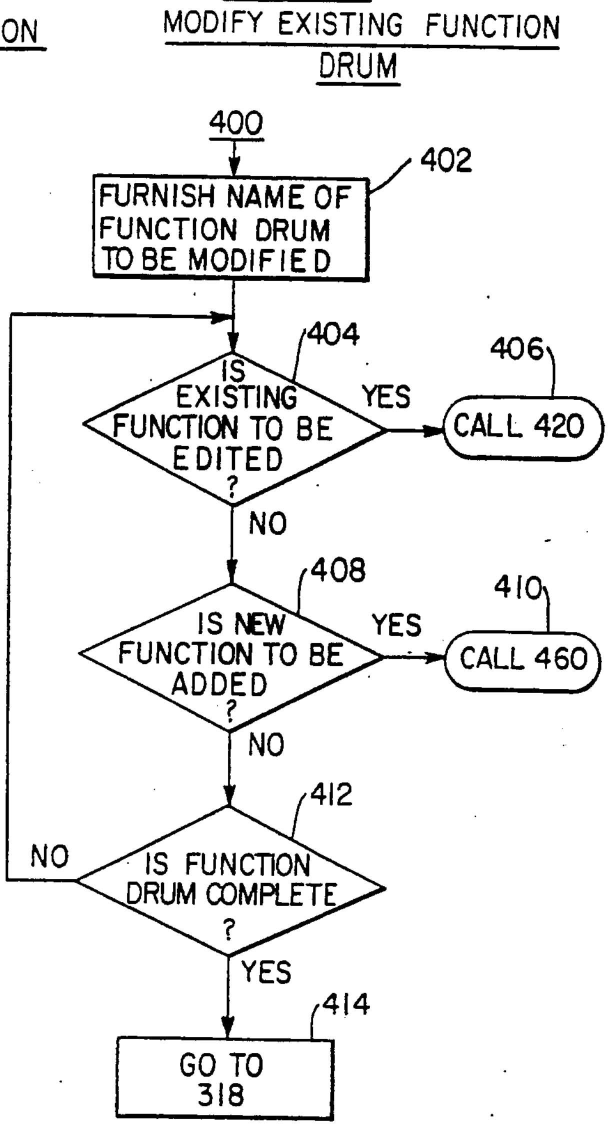
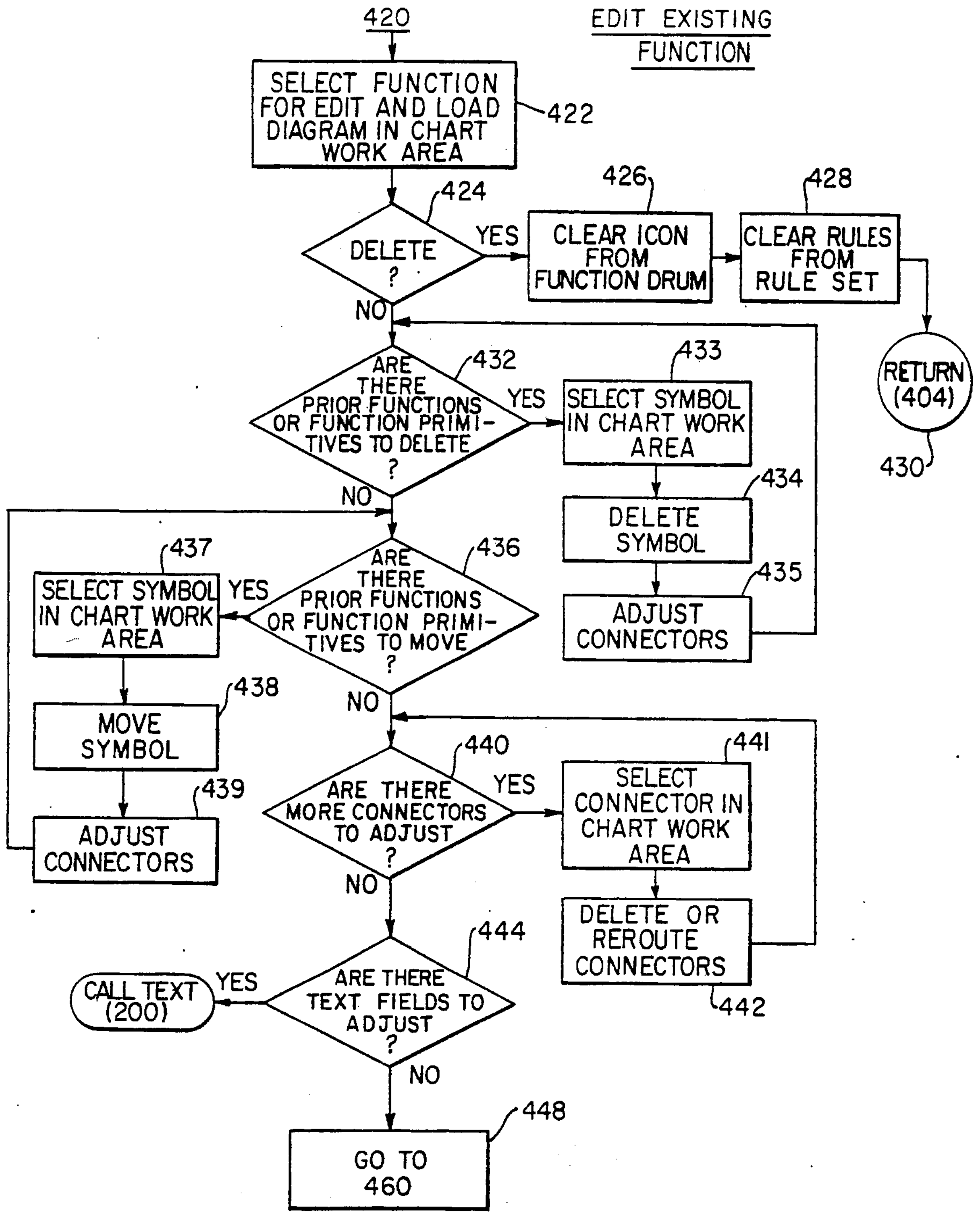

MODIFY EXISTING FUNCTION DRUM
400
FURNISH NAME OF FUNCTION DRUM TO BE MODIFIED
402
404
IS EXISTING FUNCTION TO BE EDITED ?
YES
406
CALL 420
NO
408
IS NEW FUNCTION TO BE ADDED ?
YES
410
CALL 460
NO
412
IS FUNCTION DRUM COMPLETE ?
NO
YES
414
GO TO 318

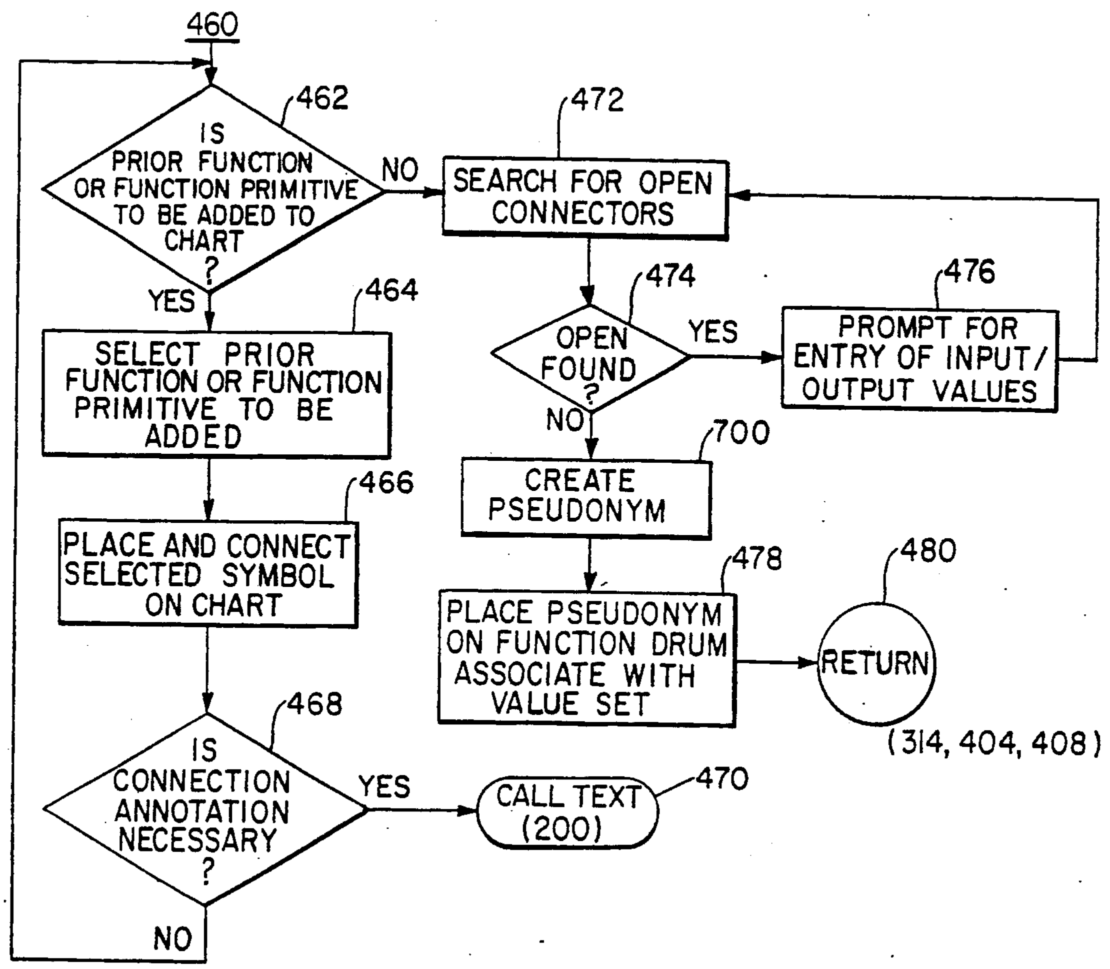
FIG. 7



420
EDIT EXISTING FUNCTION
SELECT FUNCTION FOR EDIT AND LOAD DIAGRAM IN CHART WORK AREA
422
424
DELETE ?
YES
NO
426
CLEAR ICON FROM FUNCTION DRUM
428
CLEAR RULES FROM RULE SET
RETURN (404)
430
432
ARE THERE PRIOR FUNCTIONS OR FUNCTION PRIMITIVES TO DELETE ?
YES
NO
433
SELECT SYMBOL IN CHART WORK AREA
434
DELETE SYMBOL
435
ADJUST CONNECTORS
436
ARE THERE PRIOR FUNCTIONS OR FUNCTION PRIMITIVES TO MOVE ?
YES
NO
437
SELECT SYMBOL IN CHART WORK AREA
438
MOVE SYMBOL
439
ADJUST CONNECTORS
440
ARE THERE MORE CONNECTORS TO ADJUST ?
YES
NO
441
SELECT CONNECTOR IN CHART WORK AREA
DELETE OR REROUTE CONNECTORS
442
444
ARE THERE TEXT FIELDS TO ADJUST ?
YES
NO
CALL TEXT (200)
448
GO TO 460

FIG. 8

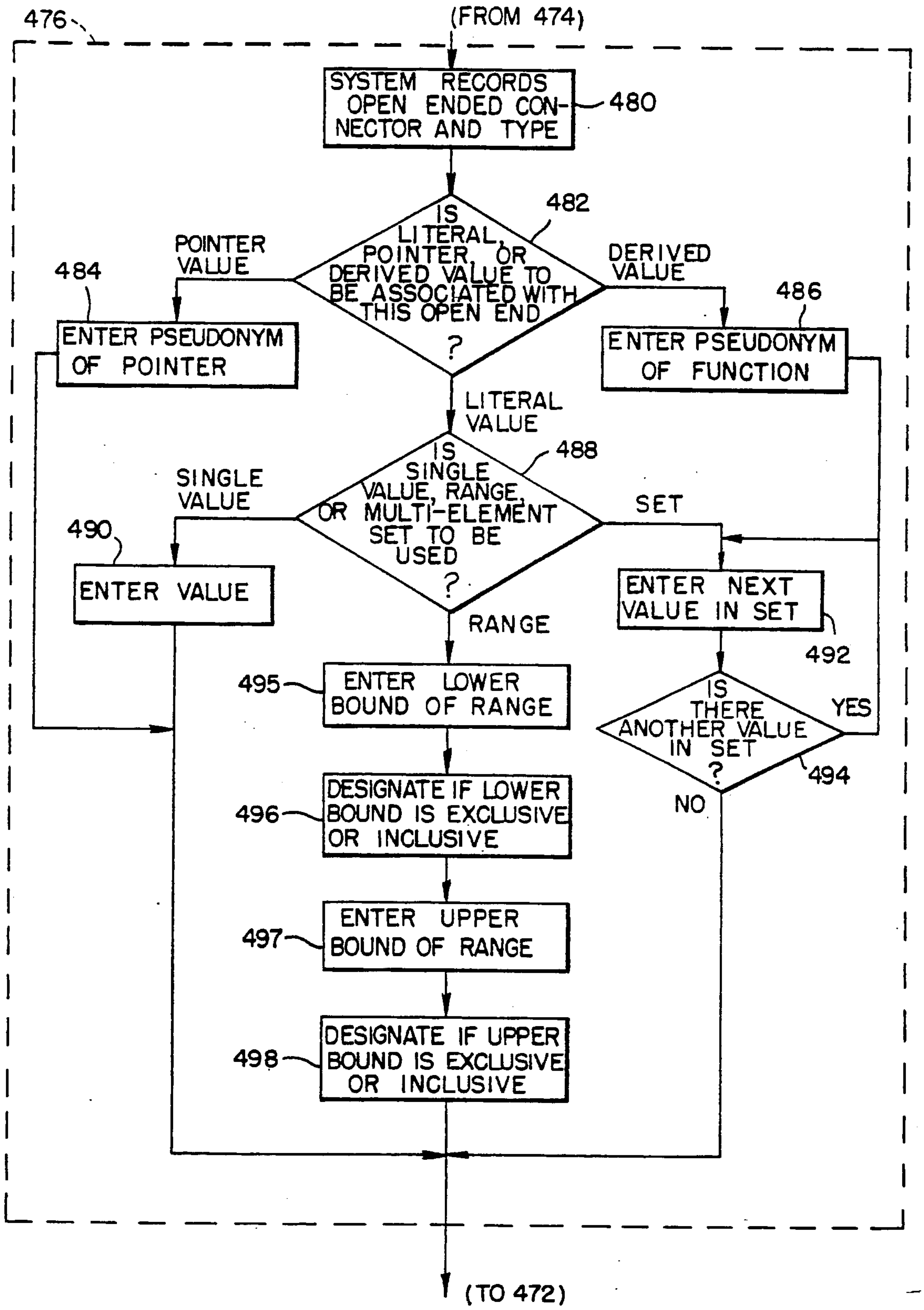
ADD NEW FUNCTION



460
462
IS PRIOR FUNCTION OR FUNCTION PRIMITIVE TO BE ADDED TO CHART ?
NO
YES
472
SEARCH FOR OPEN CONNECTORS
464
SELECT PRIOR FUNCTION OR FUNCTION PRIMITIVE TO BE ADDED
474
OPEN FOUND ?
YES
NO
476
PROMPT FOR ENTRY OF INPUT/ OUTPUT VALUES
466
PLACE AND CONNECT SELECTED SYMBOL ON CHART
700
CREATE PSEUDONYM
478
PLACE PSEUDONYM ON FUNCTION DRUM ASSOCIATE WITH VALUE SET
480
RETURN
(314, 404, 408)
468
IS CONNECTION ANNOTATION NECESSARY ?
YES
470
CALL TEXT (200)
NO

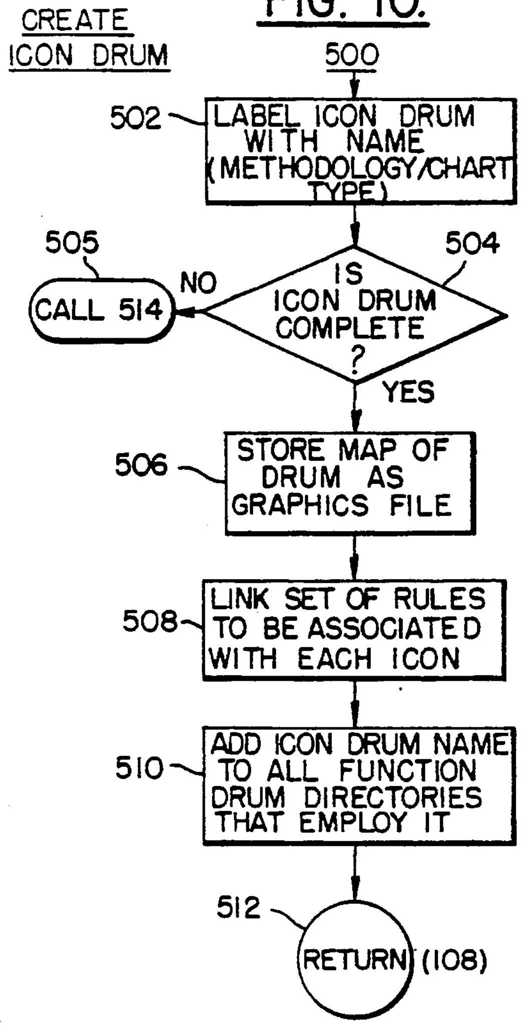
FIG. 9.

PROMPTING OF OPEN ENDED CONNECTORS


476
(FROM 474)
SYSTEM RECORDS OPEN ENDED CONNECTOR AND TYPE
480
482
IS LITERAL, POINTER, OR DERIVED VALUE TO BE ASSOCIATED WITH THIS OPEN END ?
POINTER VALUE
484
ENTER PSEUDONYM OF POINTER
DERIVED VALUE
486
ENTER PSEUDONYM OF FUNCTION
LITERAL VALUE
488
IS SINGLE VALUE, RANGE, OR MULTI-ELEMENT SET TO BE USED ?
SINGLE VALUE
490
ENTER VALUE
SET
ENTER NEXT VALUE IN SET
492
RANGE
495
ENTER LOWER BOUND OF RANGE
IS THERE ANOTHER VALUE IN SET ?
YES
494
NO
496
DESIGNATE IF LOWER BOUND IS EXCLUSIVE OR INCLUSIVE
497
ENTER UPPER BOUND OF RANGE
498
DESIGNATE IF UPPER BOUND IS EXCLUSIVE OR INCLUSIVE
(TO 472)

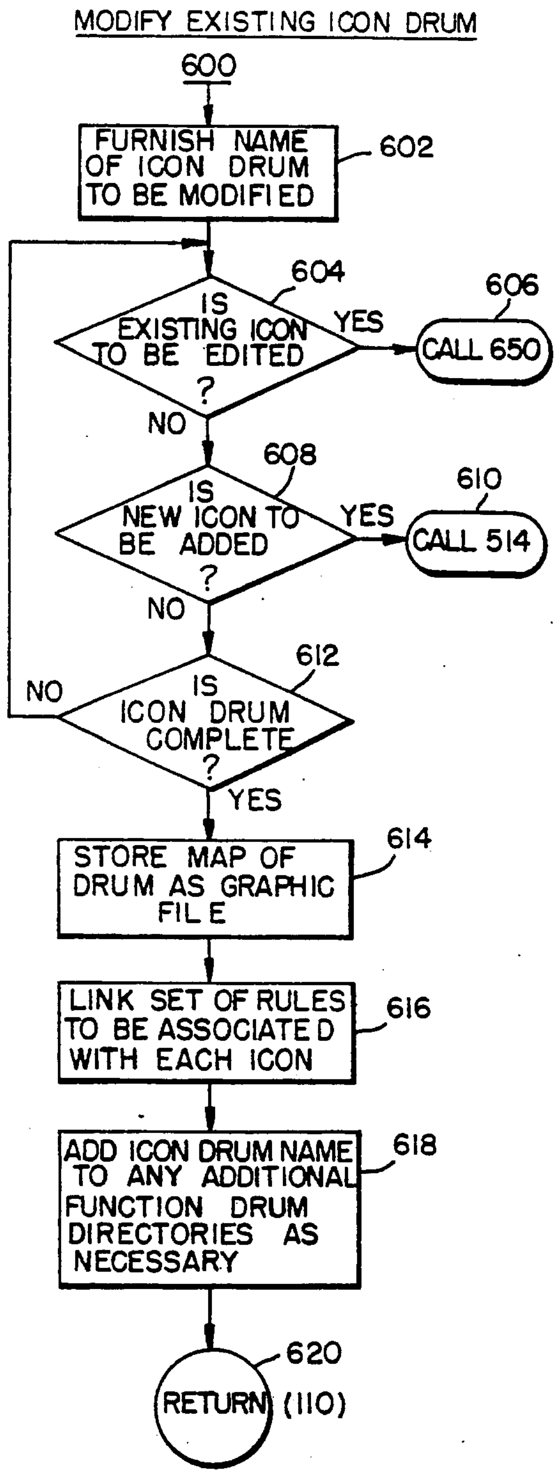
DEF017463

FIG. 10.



CREATE ICON DRUM
500
502
LABEL ICON DRUM WITH NAME (METHODOLOGY/CHART TYPE)
505
504
CALL 514
NO
IS ICON DRUM COMPLETE ?
YES
506
STORE MAP OF DRUM AS GRAPHICS FILE
508
LINK SET OF RULES TO BE ASSOCIATED WITH EACH ICON
510
ADD ICON DRUM NAME TO ALL FUNCTION DRUM DIRECTORIES THAT EMPLOY IT
512
RETURN
(108)

FIG. 12.

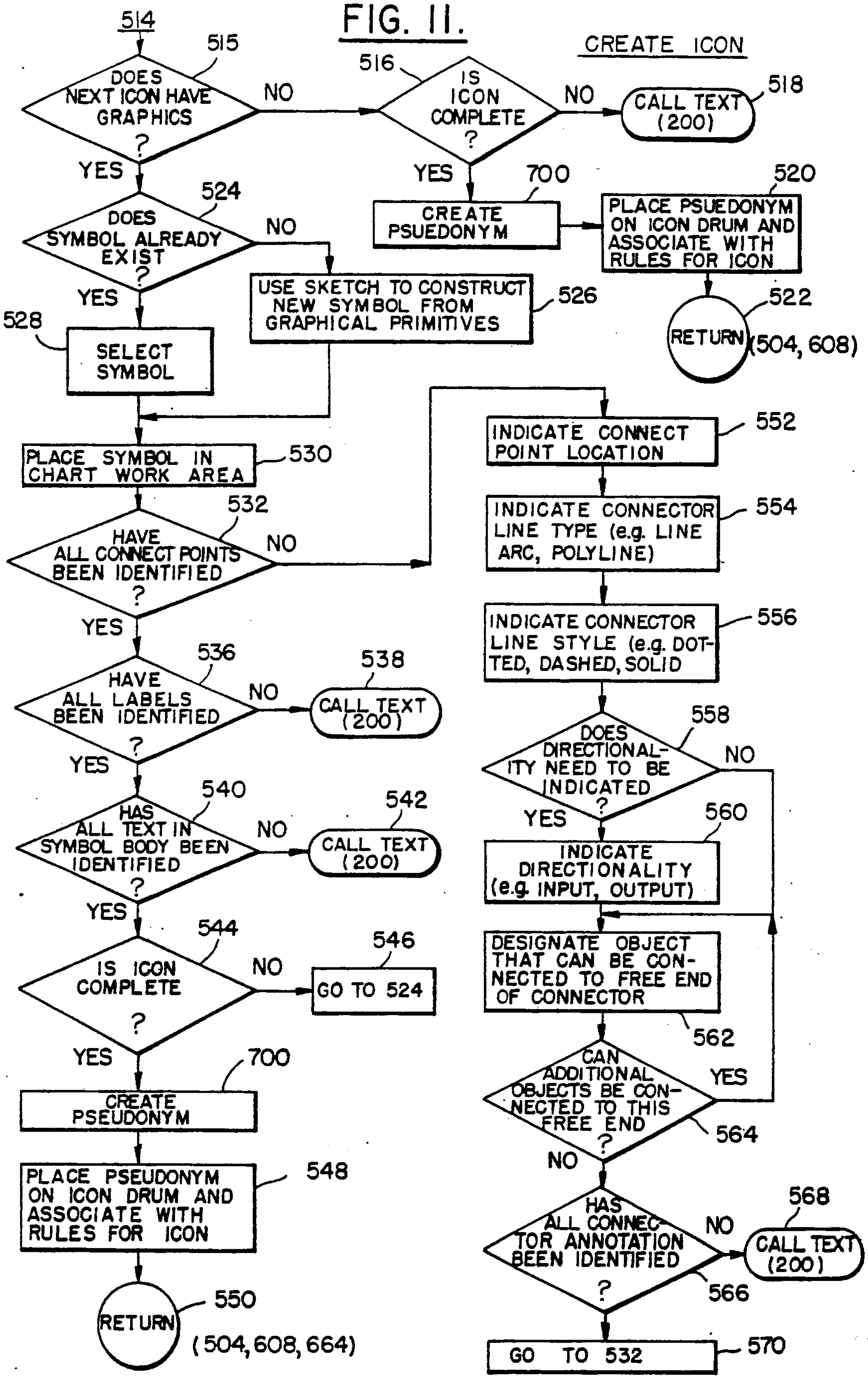


MODIFY EXISTING ICON DRUM
600
FURNISH NAME OF ICON DRUM TO BE MODIFIED
602
604
IS EXISTING ICON TO BE EDITED ?
YES
606
CALL 650
NO
608
IS NEW ICON TO BE ADDED ?
YES
610
CALL 514
NO
612
NO
IS ICON DRUM COMPLETE ?
YES
614
STORE MAP OF DRUM AS GRAPHIC FILE
616
LINK SET OF RULES TO BE ASSOCIATED WITH EACH ICON
618
ADD ICON DRUM NAME TO ANY ADDITIONAL FUNCTION DRUM DIRECTORIES AS NECESSARY
620
RETURN
(110)

DEF017464

Case 5:03-cv-04669-JW Document 243-7 Filed 09/10/2004 Page 10 of 27

FIG. 11.


CREATE ICON
514
515
DOES NEXT ICON HAVE GRAPHICS ?
NO
516
IS ICON COMPLETE ?
NO
CALL TEXT (200)
518
YES
YES
700
CREATE PSUEDONYM
520
PLACE PSUEDONYM ON ICON DRUM AND ASSOCIATE WITH RULES FOR ICON
524
DOES SYMBOL ALREADY EXIST ?
NO
USE SKETCH TO CONSTRUCT NEW SYMBOL FROM GRAPHICAL PRIMITIVES
526
RETURN
522
(504, 608)
YES
528
SELECT SYMBOL
PLACE SYMBOL IN CHART WORK AREA
530
532
HAVE ALL CONNECT POINTS BEEN IDENTIFIED ?
NO
INDICATE CONNECT POINT LOCATION
552
INDICATE CONNECTOR LINE TYPE (e.g. LINE ARC, POLYLINE)
554
INDICATE CONNECTOR LINE STYLE (e.g. DOTTED, DASHED, SOLID
556
YES
536
HAVE ALL LABELS BEEN IDENTIFIED ?
NO
538
CALL TEXT (200)
558
DOES DIRECTIONALITY NEED TO BE INDICATED ?
NO
YES
540
HAS ALL TEXT IN SYMBOL BODY BEEN IDENTIFIED ?
NO
542
CALL TEXT (200)
YES
560
INDICATE DIRECTIONALITY (e.g. INPUT, OUTPUT)
YES
544
IS ICON COMPLETE ?
NO
546
GO TO 524
DESIGNATE OBJECT THAT CAN BE CONNECTED TO FREE END OF CONNECTOR
562
YES
700
CREATE PSEUDONYM
CAN ADDITIONAL OBJECTS BE CONNECTED TO THIS FREE END ?
YES
564
NO
PLACE PSEUDONYM ON ICON DRUM AND ASSOCIATE WITH RULES FOR ICON
548
HAS ALL CONNECTOR ANNOTATION BEEN IDENTIFIED ?
NO
568
CALL TEXT (200)
566
RETURN
550
(504, 608, 664)
GO TO 532
570

DEF017465

FIG. 13.

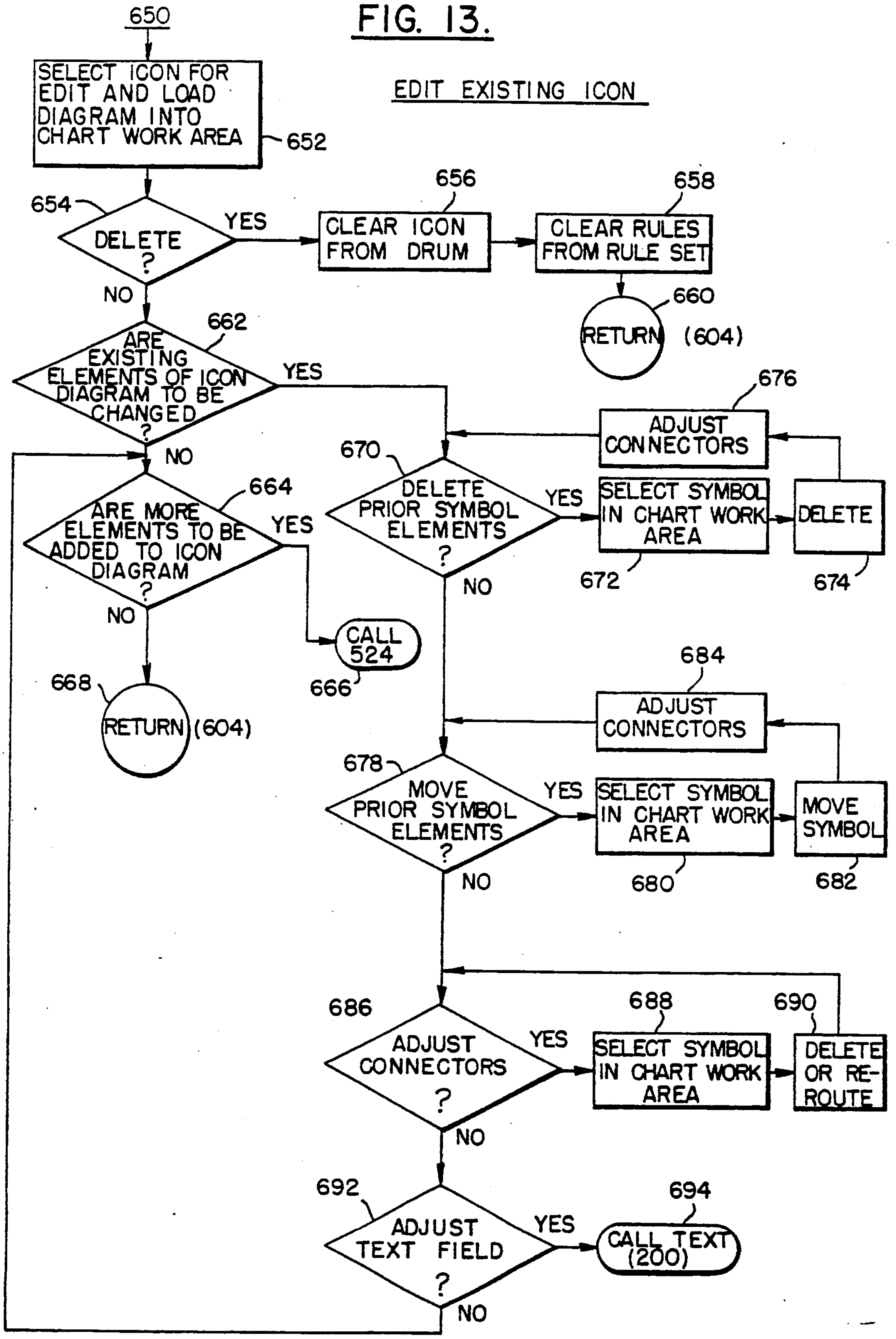

650
EDIT EXISTING ICON
SELECT ICON FOR EDIT AND LOAD DIAGRAM INTO CHART WORK AREA
652
654
DELETE ?
YES
NO
656
CLEAR ICON FROM DRUM
658
CLEAR RULES FROM RULE SET
660
RETURN (604)
662
ARE EXISTING ELEMENTS OF ICON DIAGRAM TO BE CHANGED ?
YES
NO
676
ADJUST CONNECTORS
670
DELETE PRIOR SYMBOL ELEMENTS ?
YES
SELECT SYMBOL IN CHART WORK AREA
DELETE
672
674
NO
664
ARE MORE ELEMENTS TO BE ADDED TO ICON DIAGRAM ?
YES
NO
CALL 524
666
668
RETURN (604)
684
ADJUST CONNECTORS
678
MOVE PRIOR SYMBOL ELEMENTS ?
YES
SELECT SYMBOL IN CHART WORK AREA
MOVE SYMBOL
680
682
NO
686
ADJUST CONNECTORS ?
YES
688
SELECT SYMBOL IN CHART WORK AREA
690
DELETE OR RE-ROUTE
NO
692
ADJUST TEXT FIELD ?
YES
694
CALL TEXT (200)
NO

FIG. 14.

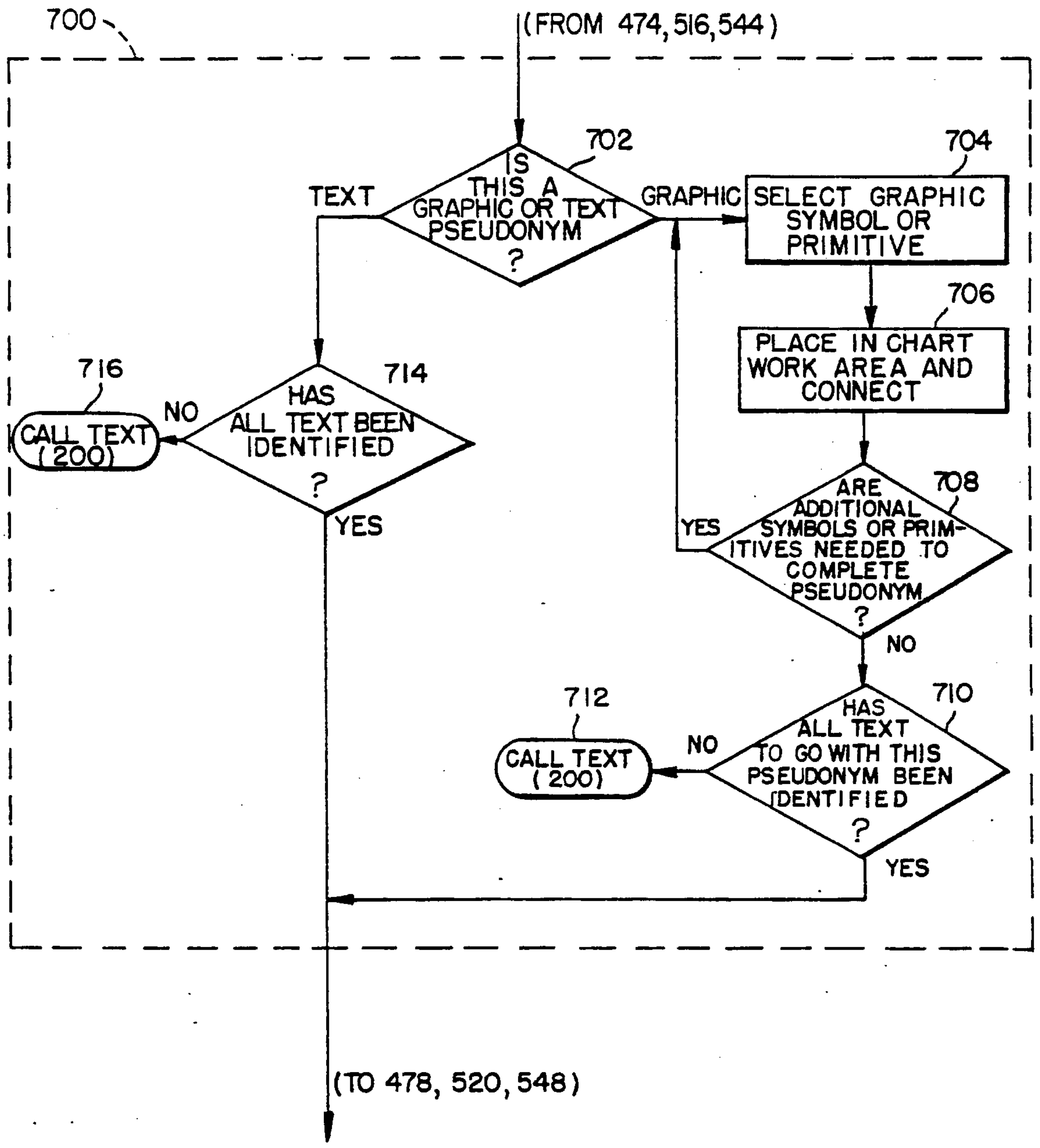

PSEUDONYM-SUBROUTINE
700
(FROM 474, 516, 544)
702
IS THIS A GRAPHIC OR TEXT PSEUDONYM ?
TEXT
GRAPHIC
704
SELECT GRAPHIC SYMBOL OR PRIMITIVE
706
PLACE IN CHART WORK AREA AND CONNECT
714
HAS ALL TEXT BEEN IDENTIFIED ?
NO
716
CALL TEXT (200)
YES
708
ARE ADDITIONAL SYMBOLS OR PRIMITIVES NEEDED TO COMPLETE PSEUDONYM ?
YES
NO
710
HAS ALL TEXT TO GO WITH THIS PSEUDONYM BEEN IDENTIFIED ?
NO
712
CALL TEXT (200)
YES
(TO 478, 520, 548)

FIG. 15.

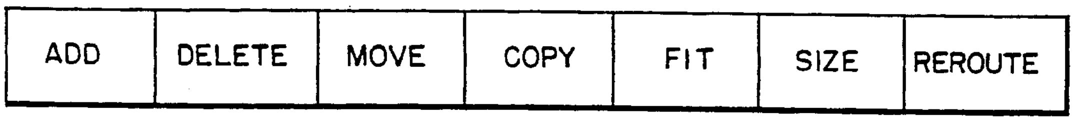
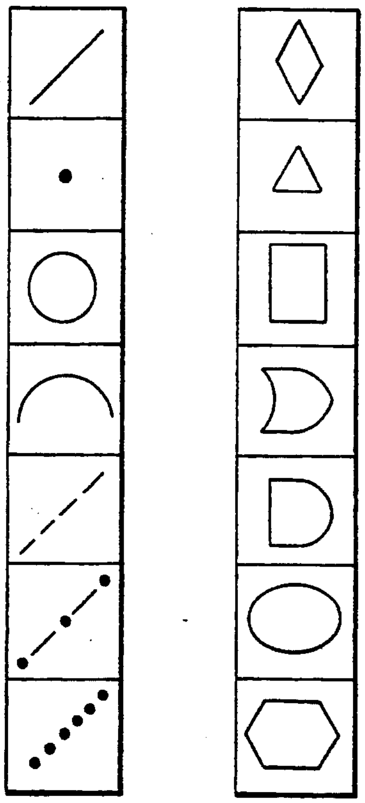


FIG. 16.



SCHEMATIC DIAGRAM GENERATING SYSTEM USING LIBRARY OF GENERAL PURPOSE INTERACTIVELY SELECTABLE GRAPHIC PRIMITIVES TO CREATE SPECIAL APPLICATIONS ICONS

The present application includes a microfiche appendix.

BACKGROUND OF THE INVENTION

The present invention relates to computer systems and more particularly to a new type of interactive computer system for enabling a problem solving user to create (i.e. plan and evaluate) solutions to problems by building schematic diagrams representative of the solutions. The system of the present invention is useful in designing solutions which lend themselves to symbolic representation, e.g. in the form of schematic diagrams such as flow charts, process diagrams, and the like. Examples of designs which utilize such solutions are the layout of assembly lines for product manufacturing, the design of computer systems, computer software design, and the design of refineries and chemical plants. The interactive computer system disclosed herein also has application to the creation and use of "smart forms", e.g. for income tax reporting purposes, which are capable of satisfying the unique requirements of any problem solving user.

Past attempts to develop computer systems which are capable of creating problem solutions have been implemented through artificial intelligence techniques. Artificial intelligence is the branch of computer science that attempts to make machines emulate intelligent behavior. There has been success in enabling a computer to reason from knowledge in a limited domain, and in some instances computer programs implementing artificial intelligence techniques can exceed human performance. Such programs use a collection of facts, rules of thumb, and other knowledge about a given field, coupled with methods of applying those rules, to make inferences. These programs have been applied to such specialized fields as medical diagnosis, mineral exploration, and oil-well log interpretation. Since such programs often must make conclusions based on incomplete or uncertain information, they differ substantially from conventional computer programs which solve problems in accordance with pre-defined algorithms and complete data sets. The power of such systems results from entering large amounts of knowledge into the computer. It is such knowledge data, and not sophisticated reasoning techniques, that is responsible for the success of such "expert" systems. An introduction to such systems is provided in the article entitled "Expert Systems: Limited But Powerful", by William B. Gervarter, *IEEE Spectrum,* August, 1983, pgs. 39–45.

One problem with knowledge based expert systems is the vast amount of data which must be entered into the computer in order to provide a useable system. The more knowledge a system is given, the better will be its solution. There is a tradeoff, however, because greater search time is required when more information is entered into the system. Other drawbacks to such systems are their narrow domain of expertise, the requirement that problem solving users describe problems in a strictly defined formal language, and the extensive training required to teach problem solving users to use such systems.

The present invention provides a general purpose machine for designing problem solutions which can be represented in schematic diagram form, and avoids many of the problems associated with prior knowledge based systems. Although the system can be used to build problem solutions in any field in which solutions can be represented in schematic form, the system will be described herein primarily with reference to the automation of the proces used by programmers, data processing analysts, and system methodology designers to create computer software for commercial applications. The description of the invention in connection with the creation of computer software is not intended to limit the scope of the invention in any manner. Those skilled in the art will appreciate that as a general purpose system, the apparatus and techniques of the present invention will have broad application to the design of problem solutions in any field where such solutions can be represented in schematic diagram form.

Current methods of software design are time consuming and unsuitable for collaboration between many people. It would be advantageous to provide tools for software creation professionals which allow them to employ techniques derived from the world of computer aided design and engineering. These techniques would be utilized to create the logical analysis of the system whose design is being sought, to plan the solution itself, to identify the parts of the solution that can be implemented as independent modules, to identify the elements of communication between such independent modules in the solution's implementation, to design the independent solution components, and to test for their logical correctness prior to actually including the design solutions in a computer program. It would also be advantageous to be able to utilize functional modules previously designed in the context of other solutions for other problems, in the solution effort for the problem at hand. Such re-utilization of previously developed modules would increase the efficiency of the design process. The present invention relates to such a system.

In the system of the present invention, problem solutions can be generated in schematic diagram form in accordance with certain methodologies. The solutions are such that information can be extracted from a design at a graphical level and be used as the basis for subsequent input to automatic code generation tools or other automated functions, such as the generation of masks for fabricating integrated circuits. The system enables problem solving users to initially sketch what they conceive to be their current notion of a possible solution strategy. This mode of system use is referred to as the "chart" mode. After the sketching process has led to some degree of satisfaction and acceptance of the solution strategy, a formal rule-based schematic of the solution can be created which logically conveys each aspect of the solution prior to its implementation. The creation of such schematics occurs at the "schematic" level of the system. The formal schematic is then reviewed by the system to verify that it is correct with regard to all of the internal formal rules, details and mechanisms of the methodology that is employed to create it.

A key function of the present invention is the provision of means for enabling a methodology designer to create a library of logically or methodologically based schematic symbols, or graphic icons, and related formal functions which govern the employment and manipulation of the graphic icons when combined into a problem solution. The icons and functions ar created by the methodology designer on an interactive basis with the system. Thus, during the creation of such icons and functions, the system prompts the methodology designer to identify, by way of example, the parameters and use of each icon or function. On the basis of these parameters, the system generates and stores a specific set of rules for each and every icon and function which completely and logically establish how each icon and function can be used to build problem solutions. In prompting a methodology designer to identify such parameters, the system requires the methodology designer to provide examples as to how the icon or function being created can be connected to other icons and/or functions. Such examples enable the system to generate the specific rules for the use of the icons and functions. Thus, the rules are built "by way of example".

As noted, the methodology designer can create new graphic icons and new formal functions. The new functions can be represented either in a text format or symbolically as "function icons". Such new functions can be built in three ways, to be explained hereinafter in detail. In general, these three methods comprise (1) concatenating function primitives already stored in the system; (2) concatenating function primitives with other existing complex functions; and (3) creating value tables for a function icon by assigning input and output values to all of the open ended connectors present on the function icon, which values are used by the system to establish a transfer function across the icon. The types of values which can be assigned when creating value tables are actual numerical values, indirect values (i.e., pointers to actual values), and inputs or outputs of other functions.

Unlike prior knowledge based systems, which require the entry and storage of a vast amount of information, the system of the present invention builds rules on the basis of procedure within a field of intent. Each such field of intent may be represented by its own library of icons and functions within the system. The system thereby precludes interference between intents. Since a methodology designer can create unique sets of icons and functions to accomplish desired intents from axiomatic primitives stored in the system, a truly general purpose machine results. Once a methodology designer creates a special library of icons and functions for a desired application (i.e., intent), the system can then accommodate any problem solving user who desires to build problem solutions for that application in schematic diagram form. The methodology designer has complete control over what applications the system can be used for and in the definition of the methodology to be used in building solutions for those applications. This is a substantial advance over prior systems, which can be used only for the creation of problem solutions according to an established methodology in the specific field in which the machine is "knowledgable".

SUMMARY OF THE INVENTION

In accordance with the present invention, an interactive rule based system is provided for generating problem solutions in schematic diagram form. The system operates and can be interfaced with on several levels. A first level of operation is engaged in by a "problem solving user", who builds schematic diagrams for applications available through existing machine functions. The first level problem solving user has an inventory of such functions, arranged in libraries or sets on a function drum, and an inventory of icons, arranged in libraries or sets on an icon drum and that together provide the building blocks necessary to use the system for an intended application.

A second, higher order level of operation is engaged in by a "methodology designer" who actually creates the libraries of functions and icons that are available to the problem solving user. It will be understood that the methodology designer and problem solving user can be the same or different persons; the role of each, however, is quite distinct. The methodology designer, through interactive use of the system, actually establishes the methodologies that govern the building of schematic diagrams by the problem solving user. The present invention is primarily concerned with this second level of operation, and in particular with the unique tools provided for use by the methodology designer.

The system includes a computer processor, a graphics terminal coupled to the processor, and means for providing a multi-portion split display on the graphics terminal. A plurality of functions and graphic and functional primitives are stored in the computer processor. Means are provided for enabling a methodology designer to employ the functions and select and arrange the graphic primitives using the graphics terminal to create a library of icons. Means operatively associated with the computer processor are provided for prompting a methodology designer to identify, by way of example, parameters which define each icon and its use. The system generates and stores a specific set of formal rules pertaining to the nature and use of each icon on the basis of the parameters identified. The stored formal rules are cross-referenced to the icon to which they pertain, so that whenever the icon is selected by a problem solving user for use in building a problem solution, the formal rules pertaining thereto will be accessed and applied.

In order to provide a friendly and consistent interface for a problem solving user, means are provided for symbolically displaying, in one portion of the split display, an icon drum comprising a set of icons from the icon library created by the methodology designer. Each such set of icons relates specifically to a category of formal schematic diagrams. Similarly, a set of functions is displayed on a function drum in another portion of the split display. Each such set of functions relates specifically to a category of operations applicable to the category of schematic diagrams which can be created from the icons displayed on the icon drum. Means are provided for enabling a problem solving user to access and select icons and functions displayed on the icon and function drums, and to build a solution to a problem by functionally arranging and coupling icons together on a chart work area portion of the split display. The system insures that the functional arrangement and coupling of icons is made strictly in accordance with the formal rules that apply to each of the icons so arranged and coupled and the functions which are employed to do so.

The system of the present invention can further comprise means for enabling a methodology designer to select and concatenate functions to each other and to icons, using the graphics terminal, to create more complex functions for display on the function drum. The complex functions can subsequently be used by a problem solving user to build problem solutions in the corresponding area of intent (i.e., for the corresponding application). Further, in order to create new functions, the system will analyze any new function icons created by a methodology designer to identify open ended connectors, and prompt the methodology designer to assign input or output values to the open ended connectors that become part of the employment rules for the later use of the function icon analyzed. The input and output values assigned by the methodology designer establish a transfer function across the function icon. The function icon and values, in combination, form a new function for display on the function drum and for subsequent use in building problem solutions.

In prompting a methodology designer to identify, by way of example, the defining parameters for the intended use of each graphic or function icon, the system requires the methodology designer to define the points of connection to each icon. Further, the methodology designer is required to indicate, for each connection point, whether the point is an input, an output, or a bidirectional port. A methodology designer is also required to indicate the connector line styles (e.g., dotted, dashed, solid line, etc.) which are permitted to be connected to each connection point, and to indicate the connector line types (e.g., line, arc, polyline, etc.) which are permitted to be connected to each point. In addition, the methodology designer is required to indicate, through system prompts, what other objects (e.g. icons or functions) are permitted to be connected, through a connector, to each connection point. Other system prompts require a methodology designer to identify whether any annotation is to be associated with a connector, and if so, what type of annotation is to be permitted. In order to complete the formal rules which define the proper use of each icon, the methodology designer is required to identify any fixed and variable text and other forms of annotation which are to be associated with each icon.

By establishing a list of formal rules for the use of each icon and function, the system is able to enforce the rules during the building of problem solutions. Thus, when a problem solving user functionally arranges and couples different icons together at the first level of system operation to create a problem solution, the system continuously verifies that all of the formal rules for the particular icons being used and functions applied are complied with.

Any input means well known in the art can be used to enable a problem solving user or a methodology designer to interface with the system. In a preferred embodiment, an input device known as a "mouse" is used to enable the selection of functions and icons and their placement on the chart work area of the graphics terminal, and a keyboard is used to enable text and numbers to be input.

The system of the present invention is an intentional system. It is a procedural rule based expert system, and not knowledge based with inference rules as are most prior artificial intelligence expert systems. By creating new icons and functions, and inputting the parameters which establish the formal rules for each icon and function, a methodology designer can customize the system for any intended purpose. Thus, a methodology designer could develop icons and functions to enable the system to be used by a problem solving user to build flow charts for computer software creation, to create process flow diagrams for chemical engineering problems, to create schematic diagrams for electrical engineering problems, or to create any other type of chart which can be used to build solutions to specific problems.

The formal rules established for the system are axiomatic and, as such, the system is closed, complete, and consistent. Such formal rules are never modified or broken to fill a special need; once established there can be no exceptions to their application when the associated icons or functions are used by a problem solving user. Thus, those skilled in the art will recognize that the system of the present invention is significantly different from any computer implemented problem solving or design system known heretofore. In particular, such prior art systems operate in accordance with heuristic techniques (i.e. analogical, allegorical, metaphorical, and paradigmatic) and not the formal logical techniques (i.e. procedural, inductive, abductive, and deductive) applied in the present system. Heuristic based systems utilize rules of thumb or empirical knowledge to guide a problem solution and are judgmental in nature. Thus, they often result in ad hoc solutions which are not entirely consistent with past results. In the present system, the establishment of and adherence to formal rules insures consistency and provides a truly general purpose machine with which a methodology designer can implement any desired intent.

Since the system enables the creation of icons and functions which are each associated with specific formal rules within an intent, and the rules are always consistently enforced at all levels of system operation, problem solutions generated by the system can easily be supported and analyzed at various levels. For example, program description information can be extracted from a completed solution and be entered into a program description file for subsequent use in executing the problem solution. Thus, in the case of software creation, the extracted program information could be used by automatic code generators to produce actual software code. In the case where masks for integrated circuits are to be created, the program description information can be used to drive mask generators.

Information which can be extracted from problem solutions generated by the system of the present invention includes data of several kinds. For example, if the problem solution generated concerns the creation of computer software, control information will be extracted which governs the operating environment required for successful execution of the program when it has been converted into executable code. Also, the formal procedural description of the logical functioning of the program will be extracted, along with the formal description of the data types of the information employed within the program. If a problem solution concerns data flows rather than program flows, equivalent extractions could be made for use in a data description file as opposed to a program description file. The extraction of data from data files for future use is a process well known to those skilled in the art. In the present instance, extraction of information can be used to support requirements analysis of problem solutions to be generated, for decision support functions, to provide functional analysis of a solution which has been created, to provide classification and reclassification of extracted properties, to enable relational analysis, and to explore new applications (new rules) for the system. While the present invention is not concerned with any of these applications per se, it is important to recognize that the logical techniques used by the present system make the future provision of such applications obtainable in a straightforward manner.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a perspective view of the components of the system of the present invention;

FIG. 2 is an example display which might appear on the graphics terminal;

FIG. 3 is a flow chart showing the operation of the "Rule Tool" which enables a methodology designer to create and modify function and icon drums containing functions and icons having rules associated therewith;

FIG. 4 is a flow chart of the Rule Tool text subroutine;

FIG. 5 is a flow chart of the procedure for creating a function drum with the Rule Tool;

FIG. 6 is a flow chart of the procedure for modifying an existing function drum with the Rule Tool;

FIG. 7 is a flow chart of the procedure for editing existing functions with the Rule Tool;

FIG. 8 is a flow chart of the procedure for adding a new function with the Rule Tool;

FIG. 9 is a flow chart of the procedure for prompting open ended connectors;

FIG. 10 is a flow chart of the procedure for creating an icon drum;

FIG. 11 is a flow chart of the procedure for creating a new icon;

FIG. 12 is a flow chart of the procedure for modifying an existing icon drum;

FIG. 13 is a flow chart of the procedure for editing an existing icon;

FIG. 14 is a flow chart of the procedure for creating pseudonyms for functions or icons;

FIG. 15 is a graphical example of function primitives which are stored in the system of the present invention; and

FIG. 16 is a graphical example of graphics primitives which are stored in the system of the present invention.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT

Turning now to FIG. 1, the system 10 of the present invention comprises a graphics terminal 12 with a display screen 14. Graphics terminal 12 is coupled to a computer processor 26, which also supports keyboard 16 and mouse 18. The graphics terminal, display screen and computer processor can be a personal computer ("PC") of a type well known in the art, such as the PC manufactured by Wang Laboratories of Lowell, Mass. Mouse 18 includes three switches 20, 22, and 24 which are assigned, for right-handed use respectively, the actions of select, cancel and evaluate. For left handed use, the position of the select and evaluate buttons can be reversed, to provide for corresponding finger use to depress a button.

The select button 20 indicates to the system, when pressed, the selection of a free space point, object, icon, or function based on the position of a cursor 29 controlled by mouse 18 on screen 14. To select a free space point, the cursor is positioned to the desired location on screen 14 and the select button 20 is actuated. A free space point is any point not occupied by, or within the tolerence of, an object. A marker is displayed at the selected point. An object is selected by positioning the cursor on or near (with a search tolerance) the object on screen 14 and actuating select button 20. An icon is selected by placing the cursor within a defined boundary box for the icon. A function is selected by simply moving mouse 18 so that the cursor on screen 14 is

within a box enclosing the function, and then actuating select button 20.

Function selection can also be associated with evaluate button 24. To perform the association, mouse 18 is moved so that cursor 29 is within the desired function box, and then evaluate button 24 is actuated. The function selected will become associated with evaluate button 24 and, from that point on, actuating the evaluate button will produce the same result as positioning the cursor over the function and actuating the select button. The advantage of the association of a function with evaluate button 24 is the minimization of eye & hand movement.

Cancel button 22, when actuated, causes the system to disregard the last action. This is useful in overcoming an unintentional actuation of one of buttons 20 or 24, or to recover when a problem solving user or methodology designer has changed his or her mind with respect to the next action to be taken.

Keyboard 16 is provided in order to enable the input of numbers and text into the system. The design and use of keyboards and mouse devices and their interface with computer systems, including graphics terminals, executive work stations, computer aided design systems and the like is well known, and those skilled in the art are aware of how to implement such input devices. Any of the known systems for interfacing mouse 18 and keyboard 16 with computer processor 26 and graphics terminal 12 can be used in connection with the present invention. It is noted that alterntive input means, for instance a graphics tablet, can be used instead of mouse 18.

FIG. 2 shows a sample multi-portion split display which can be output on screen 14 of graphics terminal 12 in accordance with the present invention. The display is "split" into system defined areas called regions. A region is defined to be a rectangular area of the screen enclosed by a border of lines. Each region serves a functional purpose, and is adjacent to another region. None of the regions overlap.

In FIG. 2, region 28 is the chart work area, and is a view of the drawing or a portion thereof that is presently being worked on by a problem solving user or methodology designer. Chart work area 28 may be thought of as a window through which one is viewing a chart that sits beneath. It is within this region that schematic diagrams are created and edited by a problem solving user, and new icons and functions are created by a methodology designer.

The entire available area onto which a chart may be created is called the drawing space. At any one time, the drawing region in chart work area 28 is a display of a part, or all, of that drawing space. Since only a portion may be visible, means is provided to view the other areas of the drawing space. Such means is provided by navigation region 58.

Navigation region 58 contains a rectangle 59 called the view port. The view port represents that portion of the chart currently visible in chart work area 28. For example, if view port 59 is half the size of the navigation region 58, and directly in the center, then exactly half of the drawing space is visible in chart work area 28. Since view port 59 is centered, the half that is visible in chart work area 28 is the center of the drawing space.

To view a different portion of the drawing space, a function which is built into the system, called the move function, is used to move the view port 59 to the desired location within navigation region 58. In order to scale the size of the drawing space, the size of view port **59** must be altered. This is done by using a size function built into the system. Once the move and/or size functions are completed, the drawing region is refreshed to display the correct portion of the chart at the correct scale.

Another region on display screen **14** is known as the icon drum region **60**, which displays a portion of what can be thought of as a band of icons upon a drum. Icons represent various symbols, lines, and text that can be selected for placement in the drawing region of chart work area **28**. The band of icons that is placed upon the drum in icon drum region **60** depends upon the level in which one is working on the system and the type of schematic diagram that is to be created or edited by a problem solving user, or the new function or icon to be created by a methodology designer. There are actually two drums within region **60** each having their own band of icons and which may be individually manipulated. One band contains icons **62**, **64**, **66**, **68**, **70** and **72** (and others not visible). The other band contains icons **74**, **76**, **78**, **80**, **82**, and **84** (and others not visible). If a desired icon is not displayed in region **60** when needed, the drums can be scrolled until the desired icon appears.

Underneath each icon drum are two pair of arrows **50**, **52** and **54**, **56**. These are scroll control arrows and represent the function of spinning the drum above the arrows selected. If, for example, arrow **50** is selected, the drum containing icons **62–72** will be spun upward. Selecting arrow **52** would spin this icon drum downward. Similarly, selecting either of arrows **54** or **56** would spin the icon drum comprising icons **74–84** upward or downward, respectively. The scroll control arrows are contained in region **44** of display screen **14**. Also contained in region **44** are scroll arrows **46** and **48** which are used to horizontally scroll the function drum **40** as described below.

Region **40** in display screen **14** is the function drum region and contains the functions of the system that are applicable to icons or schematic diagrams or their parts. As shown in FIG. **2**, the functions in region **40** are in the form of text strings. As will be described hereinbelow, however, region **40** can also contain symbols which represent functions or function primitives. If a desired system function is not displayed in region **40** when needed, right and left scrolling arrows **46** and **48** can be used to spin the function drum toward the right or toward the left until the desired function appears.

Region **42** on display screen **14** is a prompt region which serves two purposes. First, the system uses region **42** to display prompts, which are usually in the form of text strings, to the problem solving user or methodology designer. Second, any textual data entered by a problem solving user or methodology designer through keyboard **16** is echoed in prompt region **42**.

An informational region **31** on display screen **14** provides a problem solving user or methodology designer with four selectable functions presented by text strings. These functions are labeled "parameters", "clear", "exit", and "help". The selection of the parameters function **32** brings up all of the parameters which can be set by a problem solving user or a methodology designer within the system. Examples of such parameters are text height, left or right handedness, line spacing, a parameter known as drum roll which defines how many icons are scrolled when the scroll arrows are used, and

the scaling factor for defining the percentage any object is scaled when the size function is applied.

The clear function **34** in informational region **31** is equivalent to multiple cancels. If the problem solving user's or methodology designer's intent is to back up to the point of the previously committed function, the cancel function could be selected as many times as needed, or the clear function could be invoked once.

Help function **38** in informational region **31** displays information about objects or functions selected by a problem solving user or methodology designer. The information appears in a page which overlaps the drawing region in chart work area **28**. Those skilled in the art will recognize that various types of information can be provided to a problem solving user or methodology designer by accessing the help function.

Exit function **36** in informational region **31** enables a problem solving user or methodology designer to exit from his current work with the system, saves the work and the status of the session and returns the display back to an alphanumeric menu for selecting the next activity to be accomplished with the system.

Another region of display screen **14** is error region **30**. This region is used by the system to display error messages. The message displayed describes an invalid action taken by the problem solving user or methodology designer. The message will remain visible until the next action is taken, i.e., either one of buttons **20**, **22**, or **24** is actuated on mouse **18**, or a keystroke is made on keyboard **16**.

Before discussing the heart of the present invention, which is the provision of means for enabling a methodology designer to create new icons and functions with a set of rules pertaining to each new icon and function, it is necessary to explain how the basic system can be used to sketch, in graphic form, on chart work area **28** of display screen **14**. This basic level of system operation is referred to as the "sketch" mode.

The sketch mode provides a free form diagramming capability with no restrictions imposed by the system. A variety of creation and editing functions exist. The problem solving user interfaces with the system to create sketches through mouse **18**. The relative movement of mouse **18** causes a corresponding movement of a cursor **29** on chart work area **28** of display screen **14**. A problem solving user may continually move the cursor near an icon on the icon drum **60** or a function on the function drum **40** and actuate select button **20**. This will associate mouse **18** with the object (icon or function) selected. If, for example, cursor **29** is moved into the region of icon **74**, and select button **20** is actuated, a rectangle (the object of icon **74**) will become associated with mouse **18**. Mouse **18** can then be moved to drag the rectangle across chart work area **28** until a desired location is reached. Upon activation of the select button, a rectangle will be deposited at the desired location in chart work area **28**. When a function is to be used, mouse **18** is moved to place cursor **29** over the desired function in function drum **40**, and the select or evaluate button is actuated to associate the function with mouse **18**. By manipulating functions and icons with mouse **18**, and placing icons on chart work area **28** at desired locations and with desired connections, schematic diagrams can be drawn. Interaction with other regions of display screen **14** is achieved through the use of mouse **18** in a similar manner. It is noted that in the system of the preferred embodiment, data may be selected in the postfix sequence, i.e. where the function selection occurs after data selection, in the prefix mode, where function selection preceeds data selection, or in the infix mode, where function and data selection occur alternately.

Various classes of icons may appear in the icon drum 60. The majority of icons are representations of symbols, which have meaning with respect to a particular type of chart the problem solving user is building. The creation of this type of icon by a methodology designer in accordance with the present invention is discussed later on. It is noted that geometry represented by an icon may or may not be identical to the geometry actually shown in the icon drum. Thus, where the actual symbol is too complex to fit within the icon drum display portion of display screen 14, a more simplified symbol (pseudonym for the actual symbol) will be shown. However, when the icon is placed in a chart, the symbol's actual geometry will appear. Such symbols may contain geometry consisting of lines, arcs, circles, other symbols and other geometric or graphical primitives or their components, as well as text fields.

In its simplest mode of operation, the present system can be used to create a "throw-away", electronic paper form of a diagram which a problem solving user can use for discussion or initial concept purposes. This mode of operation is referred to herein as the "chart mode". Once a problem solving user decides on a solution strategy, a schematic diagram mode can be entered which will enable a problem solving user to generate an actual problem solution in terms of a formal schematic diagram.

The icon drum initially supplied with the system will contain icons for use during the chart mode and during the schematic diagram mode of operation. The icons suppled for the chart mode are those for each type of line, arc, polyline and other graphical primitives as may be supplied. These icons may be depicted by solid, dashed and potentially other line styles.

Lines are scalar objects with no direction. They have no pre-defined association with other graphic primitives, icons, diagrams, or their parts. Lines may be attached by placing the end point of the line within the tolerence of the geometry of any graphic primitive, icon, diagram, or part of a diagram.

A polyline is a multisegmented line. Polylines are considered to be one object for the purposes of the system's functions applicable to icons and diagrams or their parts.

An arc is defined by three points; namely, the first end point, a point along the arc, and the other end point, in that order.

Additional primitive graphic icons are provided for use when building schematic diagrams. These include connectors, polyectors (poly-connectors), and conarcs.

Connectors are a special class of line or arc that always attach to a symbol at a connection point. Connectors may have text fields associated with them much like symbols do. They may have direction or they may be scalar. A connector is a single line or a single arc of any style (e.g. dashed, dotted, solid line, etc.). If it is desired to attach a connector to a symbol, the connector element must attach to a connection point on the symbol, even though the selected point was only near the actual connect point. That is, there are predefined points on a symbol which may be connected to by a connector element. Thus, connectors are distinct from free form lines which may attach to a symbol's geometry for stylistic effects of annotation, but which have no

logical connection denoting a communication path between two other elements.

A polyector is a multi-segmented connector to allow the routing of a connection from one object to another where a single straight connector path would interfere with other geometry. A polyector is considered to be one object for the purposes of the system's functions applicable to icons and diagrams and their parts. A polyector is also considered to be one object in establishing logical connectivity between symbols.

A conarc is a connector arc. It is defined in the same way as a regular arc, but it connects to either the ends of connectors, polyectors, or connection points on symbols. Conarcs may also have text fields associated with them and they may have a direction.

The icon drum in the most basic form of the system of the present invention contains three text icons. One such icon is for left justified text, one for center justified text, and one for right justified text. Other, less frequently changed parameters for text, such as font style, vertical justification, and height, can be set using the parameter table function. The text icons provide the functions of adding free form text to the chart. Other forms of text, such as a "label", are specific to a symbol or connector. Such other text forms will be discussed below in connection with the formation of new icons and functions.

Regardless of the type of text or its association or lack thereof, it is always entered in the same fashion. Assuming that all text parameters are correctly defined, an origin point is selected by moving cursor 29 to a desired position on chart work area 28. This is the point used as the basis for the text justification. When the text function is selected, a text cursor appears in the prompt region 42 of display screen 14. Simultaneously, a very narrow rectangle appears in the drawing region at the origin point. As text characters are input from the keyboard, they appear in the prompt region in the machine font. Simultaneously, the rectangle in the drawing region expands horizontally to show the area the text string would cover. Depending on the justification, the rectangle would expand to the right for left justification, to the left for right justification, and equally in both directions for center justification.

Each time a terminator (e.g. carriage return or other multiple control key sequence) is entered from the keyboard, the rectangle disappears, the text font replaces it in the chart, the prompt region is cleared and the text cursor reappears at the left most character position of the prompt region awaiting another string. The rectangle then reappears at the next indicated line on which text is to be placed in the drawing region. If no more text is to be input, either another function is selected or a null line is entered. If more text than can fit in the prompt region is required on one line, the input string will scroll horizontally in the prompt region.

As noted above, the system of the present invention is operable in a chart mode and a schematic diagram mode. The invention is concerned with the schematic diagram mode, and more particularly with the provision of tools for use by a methodology designer to enable the creation of new icons and functions for use in building schematic diagrams. The icons created by the methodology designer using the tools are formal symbols within an adopted schematic diagram methodology (e.g. for the development of computer software), rather than being general purpose graphic tokens such as those used in the chart mode. These formal icons are used for the creation of schematic drawings and enable the system to accomplish interactive design rule checking within the adopted methodology. All of the formal rules and procedures that are attributed to any given step of the methodology, and which can be further attributed to a graphic symbol or token of some form, are associated with the formal icons on the icon drum of the system. In this manner, as a problem solving user selects an icon for use in placing an instance of the represented symbol in the schematic diagram of the solution, the system concurrently, and at a very high interactive rate, continuously checks to insure that the formal design rules associated with the instance of that symbol are being adhered to by the problem solving user. Thus, even though the data content (as furnished by the problem-solving user) associated with the symbol may not be exactly correct, the logical structure of the schematic as a methodological description must be precisely accurate. The schematic diagram created at this level of system use is not a "throw-away" diagram. Rather, such schematic diagrams are problem solutions which may be used, revised, analyzed, or transformed by other, higher level systems. Such opportunities are available as a result of the methodologically accurate, complete, and logically consistent schematics which are generated by the system of the present invention.

The unique and novel features of the present system will now be explained in detail, with reference to FIGS. 3 through 14 of the drawings. FIGS. 3 through 14 inclusive are flow charts which set forth the operation of system software referred to as the "rule tool". The rule tool allows a methodology designer to create new icons and functions. The new icons and functions will then be accessible to a problem solving user for use in schematic diagram creation. The definable icons are symbols and connectors. Before describing how to create such icons, it is necessary to define what the icons are.

Symbols are the basis of the schematic diagramming process. They are the major part of the schematic diagrams, and are usually of the most interest. They can be of varying complexity. At the sketch mode of system operation, they consist of just the graphics for the symbol and the icon that represents the symbol on the icon drum. No rules of use for the symbol are associated therewith at the sketch level.

At the schematic drawing mode, a given symbol comprises not only graphics but also a specified number of connection points. A connection point defines where a connector may attach and the nature of that attachment. The direction of connection is specified as being input, output, or both (bidirectional). Each connection point also defines what other symbol(s) it may be connected with. A class of symbols or named diagrams may be specified for valid connections.

Symbols for the schematic drawing mode may also contain text fields. There are two major categories of text; namely, extractable and non-extractable. Higher level programs can use these categories to know which text fields to extract. Both types of text can be defined in terms of syntax rules for the text fields. Each text field has a position and may include text that always appears in every instance of the symbol and/or text that varies from instance to instance which the problem solving user must enter every time the symbol is placed. The placement of specific text could optionally have its own syntax rules of number of characters, or a list of valid entries. Such a text field could, for example, be a volume label which always has the form: "vol=[some character string]". The text field would define "vol=" to always appear, and have the problem solving user enter the desired volume value whenever the symbol is placed.

It is also possible to create a symbol which is composed of other symbols, and the same rules will apply. Such compound symbols look the same as a simple symbol, and contain a varying number of connection points with rules on the permissable type of connections. The only difference is that the connection points may be spread over many symbols. An example of the use of a compound symbol would be in diagramming subroutines. The chart for the subroutine could be diagrammed, and then turned into a compound symbol, with the input and output rules defined by the connection point rules of any unconnected points of the subroutine diagram. That subroutine symbol could then be used in other diagrams and the proper checking would be performed.

Connectors are used to connect symbols. They have a graphical description (e.g., linear single segment, linear multi-segment, or curvilinear), and a line style of either solid, dashed, or other form. They can have an icon representation for the icon drum. They can also have extractable and non-extractable text fields as desribed above for symbols.

Turning now to FIG. 3, box 100 provides for a methodology designer's access to the rule tool. As indicated by box 102, which is connected to the flow chart of FIG. 3 by a dashed line, the sketch program described above is used in combination with the rule tool to enable a methodology designer to create new formal icons and functions. Unlike the system operation by a problem solving user (e.g. during sketch) the use of the rule tool by a methodology designer results in the creation of icons and functions which each have a specific set of rules pertaining to the use thereof.

Once a methodology designer has accessed the rule tool, several prompts are displayed in the prompt region 42 of display screen 14 to determine what activity the methodology designer desires to undertake. At box 104, the system determines if it is the intent of the methodology designer to create a new function drum. If this is the intent, then the routine entitled "create function drum" (set forth in FIG. 5) is called at box 105. Otherwise, the rule tool proceeds to box 106 which inquires whether the methodology designer wishes to modify an existing function drum. If so, the routine entitled "modify existing function drum" (set forth in FIG. 6) is called at box 107. Otherwise, the methodology designer is asked at box 108 whether a new icon drum is to be created. If a new icon drum is to be created, the routine entitled "create icon drum" (set forth in FIG. 10) is called at box 109. Otherwise, the methodology designer is prompted at box 110 to determine if modifications are to be made to an existing icon drum. If this is the case, the routine entitled "modifiy existing icon drum" (set forth in FIG. 12) will be called at box 111. Otherwise, control will pass to box 112 which asks the methodology designer if more rule tool work is to be done. If so, control passes back up to box 104 and the process repeats. Otherwise, the methodology designer exits the rule tool at box 114.

When the rule tool calls another routine, such as indicated by boxes 105, 107, 109 and 111 of FIG. 1, control immediately passes to the routine called, and returns back to the box prior to the "call" box shown in the flow chart when a corresponding "return" command is reached in the called routine. Thus, for example, if a methodology designer intends to create a new function drum, as determined at box **104** in FIG. **3**, the "create function drum" routine will be called by box **105**. Once a return from the "create function drum" routine is reached (box **322** shown in FIG. **5**), the program flow will be returned to box **104** of FIG. **3**.

In the flow charts provided in FIGS. **3** through **14**, "return" boxes are annotated with the numbers of the boxes from which the routine could have been called, and to which control should be returned, as appropriate, once the routine has been executed. Those skilled in the art will appreciate that several returns may be nested, and control will be passed from a return box back to the box which called the routine in the proper nested order.

If a methodology designer indicates at box **104** that a new function drum is to created, the "create function drum" routine (designated by numeral **300** and shown in FIG. **5**) will be accessed. As shown in FIG. **5**, this routine requires the methodology designer, at box **302**, to label the new function drum with the name of the category of operations to which the new function drum will pertain. At box **304**, the methodology designer is required to enter the name of a chart type which can be used with this function drum. Such chart type might be, for example, flow charts for computer software design, flow charts for chemical processes, electrical circuit schematics, and the like. After entering the name of one chart type, the methodology designer is required at box **306** to indicate if there is another chart type which can be used with this function drum. If so, control is returned to box **304**, and the process repeats until all of the chart types which can be used with the function drum have been identified.

When there are no more chart types to be identified, control passes to box **308** where the rule tool generates a chart type menu for the function drum being created. The chart type menu lists all of the different chart types which can be used with the function drum. The menu is stored so that it can be displayed at a future time to a problem solving user who selects this particular function drum for use in building a schematic diagram. At such time, the chart type menu will be displayed to the problem solving user so that the desired chart type can be selected.

At box **310**, the rule tool associates the chart type menu with the name of the category of operations that the function drum was labeled with in box **302**. In box **312**, the name of the category of operations is added to the system operations menu. The system operations menu is displayed to a problem solving user at the beginning of the schematic diagram mode of system operation. The problem solving user selects a category of operations, thereby defining the applicable function drum. The chart type menu associated with function drum is then displayed and the problem solving user selectes a chart type as described above.

At box **314** of the "create function drum" routine, the methodology designer is prompted to indicate if the function drum being created is complete. If the answer is no (which will be the case the first time the methodology designer is asked the question), a routine entitled "add new function" will be called as indicated at box **316**. The "add new function" routine is shown in FIG. **8**.

New functions can be created in three different ways. A brand new function can be created by concatenating function primitives stored in the system. Examples of

function primitives are the "AND", "OR", and "NOT" logic operators. Alternatively, a new composite function can be created by concatenating function primitives with other existing complex functions. "Macro-functions" can be created by concatenating a plurality of existing complex functions.

In addition to the above, there is another type of function which a methodology designer can build using the present system. These functions are, in effect, value tables for an icon or icon/function combination which has been built. Such a value table is created by identifying all connectors which exit from a given symbol, and assigning values (pointer values, derived valves, or literal values) to all non-terminated inputs and outputs. Straightforward, known techniques (e.g. calculus of variation) are used to establish a transfer function for the symbol in accordance with the assigned input and output values.

When the "add new function" routine shown in FIG. **8** is entered at **460**, box **462** requires the methodology designer to indicate whether a prior function or function primitive is to be added to the chart. If so, control passes to box **464** which requires the methodology designer to select the prior function or function primitive to be added to the chart. The selection of the prior function or function primitive will be made by using mouse **18** to select the appropriate symbol from the function drum **40** displayed on display screen **14**. The use of mouse **18** to select symbols from the function and icon drums has already been explained hereinabove.

Once the desired symbol has been selected, the methodology designer is required at box **466** to place it on the chart work area and connect it to other symbols on the chart as appropriate. Control then passes to box **468** which asks the methodology designer whether any connector annotation is necessary. If no such annotation is necessary, control reverts to box **462** and the procedure repeats until all necessary symbols have been selected, placed, and connected on the chart. In the event connector annotation is necessary, box **468** passes control to box **470** which calls a routine entitled "text".

The "text" subroutine is shown in FIG. **4**. After entering the text subroutine at **200**, the system asks the methodology designer at box **202** if there is text already present in the chart being worked on. If not, the methodology designer is required at box **205** to designate the text start point, and at box **207** to select the text justification. Otherwise, control passes from box **202** to box **204** which requires the methodology designer to find the start point of the existing text.

The rule tool next steps the methodology designer, one character position at a time, through the new text to be added (box **208**). Creation or alteration of text commences at the designated start point and the procedure is the same whether the methodology designer is creating a new text field or modifying an existing string of text. At box **208**, the methodology designer is required to decide whether the next character position is to be retained as is or altered. If it is to be retained, control passes to box **212** which merely spaces to the next character, and redirects control to box **208**. If the character is to be altered, control passes to box **214** where the methodology designer is required to indicate if the new character is to be fixed text or variable text. If fixed text is desired, control passes to box **216** which requires the methodology designer to enter the desired fixed character. Control then passes back to box **208**. If variable text is desired, control passes to box **218** which requires the methodology designer to enter the variable text designator which indicates to the system that a problem solving user will ultimately have to enter text at this character position. Control is then passed back to box 208 and the process continues until the methodology designer indicates that there is no next character position (i.e., the end of the text line has been reached). At this point, control will pass to box 220 where a line termination designator is entered. At box 222 the methodology designer will be prompted to indicate if another line of text is desired. If so, a line advance is generated at box 224 and control is passed back to box 208 so that the new line can be created. When all of the lines of text have been created, control will be passed from box 222 to box 226 which returns control back to the point from which the text subroutine was entered. In the present example, the text subroutine was called by box 470 of the "add new function" routine shown in FIG. 8. Thus, control will be passed back to box 468 shown in FIG. 8, which is the box that precedes the call text box 470.

Turning back to FIG. 8, once all prior functions and function primitives necessary to create a new function have been selected, placed on the chart work area, and connected to the chart, and all connector annotation has been accomplished, control will be passed from box 462 to box 472. At this point, the system analyzes the symbol currently on the chart work area to locate open ended connectors. When an open ended connector is found, box 474 passes control to box 476, which prompts the methodology designer to enter the appropriate input or output value for the open ended connector.

Box 476 is shown in greater detail in FIG. 9, entitled "prompting of open ended connectors". As shown in FIG. 9, when an open connector is found the system records the connector and its type as indicated at box 480. Then, control is passed to box 482 where the methodology designer is prompted to indicate if a literal, pointer, or derived value is to be associated with the open end of the connector. If a pointer value is designated, control passes to box 484 and the methodology designer is required to enter the pseudonym of the pointer. A pointer value is not a numerical value itself, but rather an address or "pointer" to a location which contains the value to be associated with the open ended connector. Once the pseudonym of the pointer has been designated, control passes from box 484 out of box 476 and on to box 472 of FIG. 8.

If, at box 482 of FIG. 9, the methodology designer indicates that a derived value is to be associated with the open ended connector, control is passed to box 486 which requires the entry of the pseudonym of the function from which the value is to be derived. From box 486, control exits box 476 and continues on to box 472 of FIG. 8.

If the methodology designer indicates at box 482 of FIG. 9 that a literal value is to be associated with the open end of the connector, control passes to box 488 where the methodology designer indicates whether a single value, range, or multi-element set of values is to be used. If a single value is indicated, control passes to box 490 where the methodology designer enters the value. Control is then passed out from box 476 and on to box 472 of FIG. 8.

If a range of values is to be associated with the open ended connector, control passes from box 488 of FIG. 9 to box 495 where the methodology designer enters the lower bound of the range. Then, at box 496, the methodology designer is prompted to designate if the lower bound indicated at box 495 is exclusive or inclusive. Control then passes to box 497 where the upper bound of the range is entered. Then, at box 498, the upper bound is designated as being exclusive or inclusive. Once the range has been entered, control passes from box 498, out from box 476, and on to box 472 of FIG. 8.

If, at box 488 of FIG. 9, the methodology designer indicates that a set of values is to be associated with the open ended connector, control passes to box 492 and the next value in the set is entered. At box 494 the system prompts the methodology designer to indicate if there is another value in the set. If so, control is passed back to box 492 and the process continues until all values in the set have been entered. Once the set is complete, control passes from box 494, out from box 476, and on to box 472 of FIG. 8.

After control is passed back to box 472 from box 476, the search for open ended connectors continues until values have been assigned to all of the open ended connectors. Then, control is passed to box 700 for the creation of a pseudonym for the new function. The pseudonym is the symbol which will be stored on the function drum for future access of the new function created. Although the pseudonym can be the entire symbol built by the methodology designer through the use of the "add new function" routine, such will only be possible if the symbol is simple and will fit on the function drum. Otherwise, an abbreviated form (i.e., pseudonym) will have to be created to identify the new function on the function drum. The creation of pseudonyms at box 700 is set forth more completely in FIG. 14, which shows box 700 in greater detail.

As shown in FIG. 14, box 702 prompts the methodology designer to determine if a graphic or text pseudonym is being created. If a graphic pseudonym is desired, control passes to box 704 where the methodology designer is required to select the graphic symbol or primitive from which the new pseudonym is to be built. Box 706 requires the methodology designer to place the selected symbol or pseudonym in the chart work area and to connect it to any other symbols or primitives already in the chart work area. At box 708, the methodology designer is required to specify if there are additional symbols or primitives needed to complete the pseudonym. If so, control returns to box 704 and the process continues until the symbol for the pseudonym is complete. Then, control is passed to box 710 where the methodology designer is required to indicate if any text which goes with the pseudonym symbol has been identified. If not, the text subroutine (numeral 200, FIG. 4) is called at box 712. After all text has been identified, control passes from box 710 out from box 700 and onto the next successive box.

If, at box 702, the methodology designer indicated that a text pseudonym was to be created, control passes to box 714 which determines if all text has been identified. If not, the text subroutine is called at box 716. Once all text has been identified, control is passed from box 714 to the next box in the flow chart.

In the present example, the next succeeding box is box 478 in FIG. 8. At this point, the pseudonym is placed on the function drum and associated with the value set applicable to the function. At box 480, control is returned to the portion of the rule tool from which the "add new function" routine was originally called. In the present example, control is returned to box **314** in FIG. **5**.

Once the function drum being created by the methodology designer is complete, control is passed from box **314** to box **318** and a map of the function drum is stored in the system as a graphics file. At box **320**, the set of rules to be associated with each function in the function drum is linked to the function drum for later recall any time the function drum is selected by a problem solving user. At box **322**, the rule tool returns to the point from which the "create function drum" routine was called which, in the present example, was box **104** of FIG. **3**.

If the methodology designer desires to modify an existing function drum, box **106** of FIG. **3** will pass control to box **107** which calls the "modify existing function drum" routine shown in FIG. **6**. From entry point **400**, this routine proceeds to box **402** and requires the methodology designer to furnish the name of the function drum to be modified. Box **404** then prompts the methodology designer to determine if an existing function on that drum is to be edited. If so, box **406** calls the "edit existing function" routine shown in FIG. **7**.

From entry point **420** in the "edit existing function" routine, control proceeds to box **424** which requires the methodology designer to select the function for edit and to load the diagram for that function in the chart work area. The diagram loaded into the chart work area can comprise function primitives, prior functions, or prior icons to be edited. Control then passes to box **422** which prompts the methodology designer to determine if the diagram loaded into the chart work area is to be merely deleted. If so, the diagram is deleted from the function drum at box **426**, and the rules associated therewith are cleared from the rule set at box **428**. Box **430** then returns to box **404** of FIG. **6** from which the "edit existing function" routine was called.

If the existing function is to be edited and not merely deleted, control is passed to box **432** which requires the methodology designer to indicate if there are specific prior functions or function primitives to be deleted from the diagram representing the function to be edited. If so, at box **433** the methodology designer is required to select the symbol of the prior function or function primitive in the chart work area to be deleted. This symbol is deleted at box **434**, and the methodology designer is required at box **435** to adjust the remaining connectors in the diagram as necessary. Control is then returned to box **432** where the process can repeat or continue on to box **436**. At box **436**, the methodology designer is required to indicate if there are prior functions or function primitives to be moved in the diagram currently in the chart work area. If so, the symbol to be moved is selected at box **437**, moved at box **438**, and necessary connector adjustments are made at box **439**. Again, the process repeats until all moves of prior functions or function primitives are completed.

Control next proceeds to box **440** which requires the methodology designer to indicate if any more connectors are to be adjusted. If so, a connector is selected in box **441**, deleted or rerouted in box **442**, and the process continues until all connectors have been adjusted. Control next passes to box **444** which determines if any text fields are to be adjusted. If so, the text subroutine is called at box **446**, the text fields are adjusted, and control is returned to box **444**. Once all text fields have been adjusted, the rule tool proceeds to enter the "add new function" routine shown in FIG. **8** and already discussed hereinabove. Upon completion of the add new

function routine, box **480** (see FIG. **8**) returns control to box **404** of FIG. **6**.

When there are no more existing functions to be edited, control is passed from box **404** of the "modify existing drum" routine to box **408** thereof, and the methodology designer is prompted to indicate if a new function is to be added to the function drum. If so, the "add new function" routine shown in FIG. **8** is called at box **410**. Once the new function has been added, box **480** of the "add new function" routine returns control to box **408** of FIG. **6**.

When no additional new functions are to be added, box **412** prompts the methodology designer to indicate if the function drum being modified is now complete. If not, control is returned to box **404** and the process described above continues. Once the function drum is complete, control is passed to box **414** and the rule tool proceeds to box **318** of the "create function drum" routine shown in FIG. **5**. At box **318**, the map of the function drum just modified is stored as a graphics file. At box **320**, the set of rules to be associated with each function on the modified function drum is linked for later retrieval with each function when the function is selected by a problem solving user operating the system in the schematic diagram mode. At box **322**, control is returned back to box **106** in the rule tool flow chart of FIG. **3**.

After any modification of existing function drums has been completed, control passes from box **106** to box **108** of the rule tool flow chart, and if a new icon drum is to be created, box **109** calls the "create icon drum" routine shown in FIG. **10**. After entering this routine at **500**, control is passed to box **502** which requires the methodology designer to label the new icon drum being created with a name. The name will be indicative of the methodology and chart type to which the icon drum relates. At box **504**, the methodology designer is prompted to indicate if the icon drum being created is complete. The first time this box is reached, the drum will not be complete and control will be passed to box **505** which calls the "create icon" routine shown in FIG. **11**.

Upon entry to the "create icon" routine at **514**, control passes to box **515** which prompts the methodology designer to indicate whether the next icon being created has graphics. If not, box **516** inquires as to whether the icon is complete. If the icon is not complete, and it has no graphics, then a text icon is being created and box **518** calls the text subroutine shown in FIG. **4**. Upon return from the text subroutine, a pseudonym is created for the completed text icon at box **700**. The creation of pseudonyms at box **700**, as further detailed in FIG. **14**, has already been discussed. The pseudonym for the icon is placed on the icon drum and associated with the rules for that icon at box **520**. Control is then returned by box **522** to box **504** of FIG. **10**.

If, at box **515** of FIG. **11**, the methodology designer indicates that the next icon to be created has graphics, control will pass to box **524** which determines whether the first symbol to be used in creating the icon already exists. If not, the sketch mode is used to construct a new symbol from graphical primitives already in existence (box **526**). Otherwise, the existing symbol is selected at box **528** and placed in the chart work area at box **530**. The rule tool then prompts the methodology designer, at box **532**, to indicate if all connect points for the symbol have been identified. If not, control passes to box **552** which requires the methodology designer to indicate a connect point location. The connector line type (e.g. line, arc, polyline) and connector line style (e.g. dotted, dashed, solid) are indicated at boxes **554** and **556** respectively. At box **558** the methodology designer states whether directionality has to be indicated. If so, the directionality (e.g. input, output) is indicated at box **560** and control proceeds to box **562** which requires the methodology designer to designate the object that can be connected to the free end of the connector. An object can be a symbol, function, or another schematic diagram. At box **564** the methodology designer states whether additional objects can be connected to the free end of the connector. If so, control returns to box **562** and the process continues until all of the objects that can be connected to the connector free end have been designated. Then control passes to box **566** which determines if all connector annotation has been identified. If not, box **568** calls the text subroutine of FIG. **4**. Once all connector annotation has been identified, control returns, via box **570**, to box **532** of the "create icon" routine.

Once all of the connect points for the icon being created have been identified, box **532** passes control to box **536** for the identification of icon labels. Labels are searchable text fields associated with an icon. In creating labels, the rule tool sets a flag for each label to indicate that it is a searchable item. If labels are to be identified, box **538** calls the text subroutine. Once all labels have been identified, control passes to box **540** to determine if all text in the icon symbol body has been identified. If not, the text subroutine is called at box **542**. Once all symbol body text has been identified, control passes to box **544** where the methodology designer is prompted to indicate whether the icon created is complete. If the icon is not complete, control is returned to box **524** for further building of the icon. Otherwise, a pseudonym is created for the icon at box **700** of FIG. **11**, and the pseudonym is placed on the icon drum and associated with the rules for the icon at box **548**. Control is then returned via box **550** to box **504** of the "create icon drum" routine shown in FIG. **10**.

Turning now to FIG. **10**, once the icon drum is complete, box **504** passes control to box **506** where a map of the complete icon drum is stored as a graphics file. At box **508**, the set of rules to be associated with each icon in the icon drum is stored and linked to the icons. At box **510**, the name of the icon drum just created is added to all of the directories for the function drums that can employ the icon drum. Box **512** then returns control to box **108** of the rule tool flow chart of FIG. **3**.

Once any new icon drums have been created by the methodology designer, the rule tool proceeds to box **110** so that existing icon drums can be modified. If such modifications are to be made, box **111** calls the "modify existing icon drum" routine shown in FIG. **12**.

After entering the "modify existing icon drum" routine at **600**, the methodology designer is prompted at box **602** to furnish the name of the icon drum to be modified. The designated name must, of course, be one for an existing icon drum. Once the icon drum to be modified has been identified, control proceeds to box **604** and the methodology designer is asked whether an existing icon is to be edited. If so, control passes to box **606** and the "edit existing icon" routine shown in FIG. **13** is called.

Turning to FIG. **13**, upon entry to the "edit existing icon" routine at **650**, control passes to box **652** and the methodology designer is instructed to select the icon to be edited and to load the diagram for that icon into the

chart work area. Control then proceeds to box **654** and the methodology designer is asked whether it is desired to delete the icon which has been loaded into the chart work area. If so, control passes to box **656** and the icon to be deleted is cleared from the icon drum. At box **658**, the rules associated with the deleted icon are cleared from the set of rules pertaining to the icon drum. Box **660** then returns control to box **604** of the "modify existing icon drum" routine shown in FIG. **12**.

If, at box **654**, the methodology designer indicates that the icon is not to be deleted, then control passes to box **662** which prompts the methodology designer to determine if existing elements of the icon diagram present in the chart work area are to be changed. If so, control passes to box **670** where the methodology designer is required to indicate if prior symbol elements are to be deleted. If deletion of symbol elements is desired, control passes to box **672** and the methodology designer selects the symbol to be deleted from the chart work area. At box **674**, the symbol is deleted, and at box **676** any necessary connector adjustments are made to the remaining portions of the icon. This loop continues until all necessary prior symbol elements have been deleted

Once the methodology designer is finished deleting any prior symbol elements, control passes to box **678** and the methodology designer is prompted to indicate if he intends to move prior symbol elements. If so, control passes to boxes **680**, **682**, and **684** where the symbol elements to be moved are selected, moved, and connectors are adjusted as necessary. This loop will continue until all necessary movement of prior symbol elements has been completed.

After moving any prior symbol elements, control passes to box **686** so that connector adjustments can be made. At box **688**, the symbol for a connector to be adjusted is selected in the chart work area, and at box **690** the connector is deleted or rerouted. When all connectors have been adjusted, control passes to box **692** where text fields are adjusted. The adjustment of text fields is accomplished at box **694** by calling the text subroutine shown in FIG. **4**.

Once necessary text field adjustments have been made, control passes to box **664** where the methodology designer is prompted to determine if more elements are to be added to the icon diagram. Box **664** is also accessed if, at box **662**, the methodology designer indicates that existing elements of the icon diagram are not to be changed. If the methodology designer indicates at box **664** that more elements are to be added to the icon being worked on, then control passes to box **666** and the "create icon" routine shown in FIG. **11** is accessed at box **524**. The "create icon" subroutine has already been discussed. Once any new elements have been added to the icon diagram, the "create icon" routine returns control to box **664**. Then, box **668** returns control to box **604** of FIG. **12**.

Turning back to FIG. **12**, after a methodology designer has attended to any existing icons to be edited, control passes to box **608** where the rule tool prompts the methodology designer to determine if a new icon is to be added. If so, control passes to box **610** and the "create icon" routine shown in FIG. **11** is entered at **514**. After any new icons have been added to the icon drum being modified, control is passed to box **612** and the methodology designer is prompted to indicate if the icon drum being modified is complete. If not, control is passed back to box **604** and the steps discussed above are repeated.

If the icon drum is complete, control passes from box **612** to box **614** where a map of the modified icon drum is stored as a graphics file. Then, at box **616** the set of rules to be associated with each icon in the modified icon drum is linked with the icon drum and stored. At box **618**, the name of the modified icon drum is added to any additional function drums with which the icon drum can be used, as necessary. At box **620**, control is returned from the "modify existing icon drum" routine to box **110** of FIG. **3**.

After any modification of existing icon drums, control is passed from box **110** in FIG. **3** to box **112** where the methodology designer is asked if any more rule tool work is to be completed. If so, control is returned to box **104** so that additional rule tool work can be accomplished. Otherwise, control is passed to box **114** and the rule tool is exited.

FIG. **15** depicts a set of functions and function primitives which can be supplied with the system to enable a methodology designer to build more complex functions as necessary. These functions are: ADD, DELETE, MOVE, COPY, FIT, SIZE, and REROUTE. Other function primitives, such as the logical "AND", "OR", and "NOT" functions could also be provided.

FIG. **16** depicts an icon drum containing various graphic primitives and icons which might be supplied with a basic system in accordance with the present invention. The graphic primitives and icons suppled will enable a methodology designer to create the more complex icons necessary for use by a problem solving user in generating problem solutions. The graphicl primitives and icons shown are: SOLID LINE, POINT, CIRCLE, ARC, DASHED LINE, DOT/-DASH LINE, DOTTED LINE, DIAMOND, TRIANGLE, RECTANGLE, OR GATE, AND GATE, OVAL, and ELONGATED HEXAGON.

Once a methodology designer has built the set of functions and icons which are necessary to enable a problem solving user to diagram problem solutions for the application in which the problem solving user is interested, schematic drawing of problem solutions can commence. An important feature of the present invention is that during the schematic drawing process, design rule checking can be employed on a continuous basis to assure that the problem solution diagram being generated complies with all rules generated by use of the rule tool. This feature can be accomplished because the rules established for each icon or function are stored in the system, and are always retrieved along with the associated icon or function when the icon or function is selected.

The mechanisms of the schematic drawing process are precisely the same at the level of graphic arts as are those used during the sketch mode of system operation. The use of the system in the sketch mode has already been explained. The difference between the sketch mode and the schematic drawing mode is that in the former, the icons were merely general purpose graphic tokens, whereas in the latter, the icons are formal symbols within an adopted design methodology. Thus, in the schematic drawing mode all of the formal rules and procedures that are attributed to any given step of the methodology, and which can be further attributed to a graphical symbol or token of some form, will be associated with the icon on the display screen of the graphics terminal. In that way, as the problem solving user selects an icon in order to place an instance in the schematic diagram of the solution, the system in parallel, concurrently, and at a very high interactive rate, continuously checks to insure that the design rules associated with the instance of that symbol are adhered to. Thus, the logical structure of any schematic drawing created by the problem solving user as a proposed problem solution must be precisely accurate.

It will now be appreciated that the system of the present invention enables the generation of problem solutions in schematic diagram form wherein the schematic diagrams are inherently methodologically accurate. Once such a schematic diagram has been generated, it can be analyzed, classified, and transformed by other automated systems in a consistent manner to provide desired end results. Among the functions carried out by the system on the formal schematic diagram are the verification and extraction of data to ensure that all required data entry positions and all elements of the schematic diagram are in fact completed. Further, where elements of a design are characterized across several diagrams, and where those diagrams may relate to each other along connecting points thereof, then the references between the diagrams and the contexts imposed by the diagrams can be further verified and checked to insure consistency, and to insure correctness at the level of the referencing information placed in the diagram by a problem solving user. When it has been determined that a schematic diagram is complete, consistent, and internally correct with regard to the parameters and labels employed across several diagrams, then data can be extracted as a means to analyze individual schematics and the set of schematics, as a collection, to provide program description information which is entered into a program description file for subsequent use by other automated systems, such as automatic code generators, integrated circuit mask generators, etc.

It should now be understood that the present invention provides an interactive rule based system for generating problem solutions in schematic diagram form. Schematic diagrams are built from icons (graphic primitives or complex symbols) and functions (which can also comprise icons). Each icon and function to be used in building a schematic diagram representative of a problem solution has certain rules associated therewith. These rules insure that the logical structure of the schematic diagram, as a methodological description, is precisely accurate. By enforcing the rules of use for each icon and function, the system prevents a problem solving user from designing a problem solution which is inconsistent or in violation of the methodology rules. Rule enforcement is accomplished through appropriate prompts by the system during the building of a schematic diagram, and through the use of appropriate error messages when a probelm solving user attempts to take an action which would break a rule.

The system further provides opportunity for a methodology designer to create a library of icons and functions tailored to the specific application in which that methodology designer is interested. Thus, while the system is general purpose, it can be converted by a methodology designer to a dedicated tool for generating problem solutions in a specific field.

An object code listing of the software which implements the sketch mode of the present system is appended hereto as a microfiche appendix. The object code shown is a hexadecimal dump of the executable program code, and can be run on a Wang Laboratories personal computer (Wang PC) with 640K bytes of memory, a 10 megabyte Winchester disk, and a mouse device manufactured by Display Interface Corporation of Milford, Conn. under the trademark "HiFi Mouse". The Wang PC must include the Microsoft "MS DOS Version 2.01" 1 operating system and Wang's "BIOS Version 1.21" software.

Although the present invention has been described in connection with a preferred embodiment thereof, many variations and modifications could be made. It is intended to cover all of the variations and modifications which fall within the scope of the present invention, as recited in the following claims.

I claim:

1. A general purpose interactive rule based system for generating problem solutions in schematic diagram form comprising:

a computer processor;

a graphics terminal coupled to said processor;

means for providing a multi-portion split display on said graphics terminal;

a plurality of functions and graphic primitives stored in said computer processor;

means for creating a library of icons for an intended application by enabling a methodology designer to:

select and arrange said graphic primitives using said graphics terminal, and

identify, by way of example, parameters for using each icon in response to prompts initiated by said computer processor;

means for generating and storing a specific set of rules pertaining to the use of each icon on the basis of the parameters identified;

means for symbolically displaying in one portion of said split display a set of icons from said library;

means for displaying in another portion of said split display a set of said functions; and

means for enabling a problem solving user to access and select displayed icons and functions, and to build a solution to a problem by using functions to graphically couple icons together on a chart work area portion of said split display in accordance with said rules.

2. The system of claim 1 further comprising means for enabling a methodology designer to select and concatenate functions to each other and to icons, using said graphics terminal, to create more complex functions for display and for use in building problem solutions.

3. The system of claim 1 further comprising:

means for analyzing icons created by a methodology designer to identify open ended connectors having no values assigned thereto; and

means for prompting a methodology designer to assign input or output values to said open ended connectors;

whereby said input and output values establish a transfer function across said icon, said icon and values in combination forming a new function for display and for use in building problem solutions.

4. The system of claim 1 wherein said prompts initiated by said computer processor instruct a methodology designer to define points of connection to each icon.

5. The system of claim 4 wherein said prompts instruct a methodology designer to indicate, for each connection point, whether the point is an input, an output, or bidirectional.

6. The system of claim 4 wherein said prompts instruct a methodology designer to indicate for each connection point, the connector line styles which are permitted to be connected to that point.

7. The system of claim 4 wherein said prompts instruct a methodology designer to indicate, for each connection point, the connector line types which are permitted to be connected to that point.

8. The system of claim 4 wherein said prompts instruct a methodology designer to indicate, for each connection point, what other icons or functions are permitted to be connected, through a connector, to that point.

9. The system of claim 4 wherein said prompts instruct a methodology designer to indicate, for each connection point:

the connector line types which are permitted to be connected to that point;

the connector line styles which are permitted to be connected to that point;

whether the point is an input, an output, or bidirectional;

what other icons or functions are permitted to be connected, through a connector, to that point; and

whether and what type of annotation is to be associated with a connector connected to that point.

10. The system of claim 1 wherein said prompts initiated by said computer processor instruct a user to identify any fixed and variable text and labels to be associated with each icon.

11. The system of claim 1 further comprising means for verifying that all of said rules are complied with during the building of a problem solution.

12. The system of claim 1 further comprising a mouse input device and a keyboard to enable a methodology designer or problem solving user to interface with the graphics terminal.

13. An interactive system for building schematic diagrams in accordance with formal rules and procedures comprising:

a computer;

a graphics terminal coupled to said computer;

a plurality of general purpose user selectable functions and graphic icons stored in said computer;

means for storing sets of formal rules pertaining to the use of each general purpose function and icon;

means for using said graphics terminal to create a library of special purpose icons for an intended application;

means for generating and storing a set of formal rules pertaining to the use of each special purpose icon in said library of icons;

means for accessing the set of formal rules for a particular function or icon upon selection of that function or icon by a problem solving user;

means for enabling a problem solving user to build a schematic diagram by graphically coupling selected icons together on said graphics terminal; and

means for enforcing the formal rules for each selected icon and function, during the building of a schematic diagram, by prompting a user to input any textual, numerical, or graphic data required by said formal rules and by providing an error message if an action is attempted that would break a rule.

14. The system of claim 13 further comprising:

means for selecting and concatenating functions to each other and to icons to create more complex functions for use by a problem solving user in building schematic diagrams.

15. The system of claim **14** further comprising means for:

(a) analyzing icons to identify open ended connectors having no values assigned thereto; and

(b) prompting a methodology designer to assign input or output values to said open ended connectors in accordance with said formal rules,

whereby said input and output values establish a transfer function across said icon, said icon and values in combination forming a new function for use in building schematic diagrams.

16. The system of claim **13** wherein said formal rules for each special purpose icon are generated on the basis of responses to prompting means instructing a methodology designer to define points of connection to each icon.

17. The system of claim **16** wherein said prompting means instructs a methodology designer to indicate, for each connection point, whether the point is an input, an output, or bidirectional.

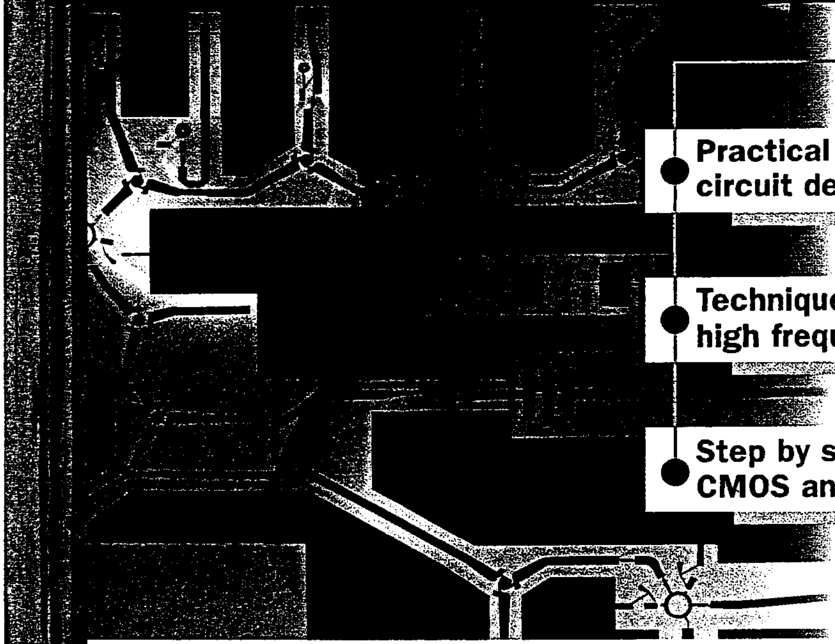
18. The system of claim **17** wherein said prompting means instructs a methodology designer to indicate, for each connection point, what other icons or functions are permitted to be coupled to that point.

19. The system of claim **18** wherein said prompting means instructs a methodology designer to identify any fixed and variable text and labels to be associated with each icon.

* * * * *

Case 5:03-cv-04669-JW Document 243-8 Filed 09/10/2004 Page 1 of 27

McGraw-Hill PROFESSIONAL ENGINEERING



- Practical layout reference for circuit designers and mask designers
- Techniques for matching, noise, high frequency issues and more
- Step by step case studies detailing CMOS and bipolar RFIC

IC Mask Design

ESSENTIAL LAYOUT TECHNIQUES

Christopher Saint / Judy Saint

DEF085303

Case 5:03-cv-04669-JW Document 243-8 Filed 09/10/2004 Page 2 of 27

Cataloging-in-Publication Data is on file with the Library of Congress

McGraw-Hill

A Division of The McGraw-Hill Companies

Copyright © 2002 by Christopher Saint and Judy Saint. All rights reserved. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher.

1 2 3 4 5 6 7 8 9 0 DOC/DOC 0 8 7 6 5 4 3 2

ISBN 0-07-138996-2

The sponsoring editor for this book was Stephen S. Chapman, the editing supervisor was Stephen M. Smith, and the production supervisor was Sherri Souffrance. It was set in Times Roman by TopDesk Publishers' Group.

Printed and bound by R. R. Donnelley & Sons Company.

McGraw-Hill books are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. For more information, please write to the Director of Special Sales, McGraw-Hill Professional, Two Penn Plaza, New York, NY 10121-2298. Or contact your local bookstore.

This book is printed on recycled, acid-free paper containing a minimum of 50% recycled, de-inked fiber.

Information contained in this work has been obtained by The McGraw-Hill Companies, Inc. ("McGraw-Hill") from sources believed to be reliable. However, neither McGraw-Hill nor its authors guarantee the accuracy or completeness of any information published herein and neither McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

DEF085304

IC Mask Design

Essential Layout Techniques

Christopher Saint

Judy Saint

McGraw-Hill

New York | Chicago | San Francisco | Lisbon | London | Madrid
Mexico City | Milan | New Delhi | San Juan | Seoul
Singapore | Sydney | Toronto

CHAPTER 1

Digital Layout

Chapter Preview

Here's what you're going to see in this chapter:

- Close look at automated layout software
- Why automated layout only works with certain cells
- Knowing the circuit really does what it should
- How to know in advance if your floorplan choice is good
- Automated programs getting stuck
- Troubleshooting tips
- Which nets to wire first
- Which nets to wire by hand
- Techniques to guarantee rule-perfect layout
- Flowchart of digital layout procedures
- Lots of feedback loops
- How to keep the power moving through big cells
- Chicken or egg wiring and timing circle
- Did you really build what you designed?
- How to build quickie chips for testing

Opening Thoughts on Digital Layout

The majority of integrated circuits built today are large. I mean really huge CMOS digital chips. One chip might have literally millions of transistors in it. It's beyond any single mask designer's capabilities to lay out a chip like that by hand—in any reasonable time frame, at least. Consequently, the majority of large digital chips are laid out with the assistance of computer-aided tools.

Understanding how these automated digital layout tools operate allows you to develop skillful daily habits in your work—even in your analog work. If you understand how the software operates, you can lay out better circuits faster, compensate for software inadequacies, and steer clear of roadblocks before they happen.

Design Process

Let's build a digital chip. In this chapter, we will follow a design team as they progress from concept, through circuit testing, and finally to the actual gate placement and wiring of a digital chip, using a suite of software tools.

Let's start. It's the circuit designer's move first.

Verifying the Circuitry Logic

Circuit designers typically use languages called **VHDL** or **Verilog** to design their enormous digital circuits. VHDL stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language, an IEEE standard since 1987. Verilog is another proprietary logic description language. We will use VHDL in our examples.

Circuit designers use the VHDL language to create a chip that exists first as only a database of numbers. The circuit designer's VHDL files are very C-like.[1] The files essentially say, for example, "I want a circuit function that adds two 16-bit numbers together." In this way, the VHDL files describe our microprocessor, our digital functions, or whatever functions we need.

These VHDL data files are then submitted to a computer simulator, which tests the chip circuitry while it is still in software form. The logic functions of the VHDL code run very quickly, much faster than a traditional transistor level SPICE simulation (but not as fast as the real silicon.)

The VHDL simulator needs to have process-specific software descriptions of each logic function it wants to use, such as rise time, fall time, gate propagation delays. This information, as well as other device parameters, is stored as a series of files that the VHDL simulator can access. Along with these electrical descriptions, there are also physical representations of each of the gates that the simulator and logic synthesizer can use. All of these files are collectively known as a **standard cell library** or **logic library.**

[1] The computer language, C.

VHDL Code Segment

```
architecture STRUCTURE of TEST is
 component and2x
 port(A,B,C,D: in std_ulogic := '1';
 Y: out std_ulogic);
 end component;
 constant VCC: std_ulogic := '1';
 signal T,Q: std_ulogic_vector(4 downto 0);
begin
T(0) <= VCC;
 A1: and2x port map(A=>Q(0), B=>Q(1),
Y=>T(2));
 A2: and2x port map(A=>Q(0), B=>Q(1),
C=>Q(2), D=>Q(3), Y=>T(4));
 Count <= Q;
```

The company that is supplying your silicon usually provides a standard cell library. Theoretically, you are given a library, which is perfect, and will stay perfect. However, updates to the library can occur quite frequently. Changes to the library can cause a once-perfect chip to stop functioning, especially if mistakes have been made in the updates. Updating a library mid-project is usually a bad move.

By looking at the results of these VHDL simulations, we can make adjustments to the circuitry before we commit the chip to actual silicon. This is a great saving.

Compiling a Netlist

Once the circuit designer has finished verifying his logic design, he will put his VHDL code through a **silicon compiler** or **logic synthesizer.** The compiler translates the high level C-like code into a file that contains all the required logic functions, as well as how they are to be connected to each other.

The file basically says, "In order to add two 16-bit numbers together, I need 25 gates and here's how they should be connected." In this way, all our logic functions are created and cross-referenced.

At this point, we know what gates we need, and we know how they must be eventually wired to each other. This file, called a **netlist,** will drive your automated layout tools.

Netlist Segment

```
module test ( in1, in2, out1);
 input in1,in2;
 output out1;
 wire \net1 , \net2 , \net3 ;
 AND2_2X U1 ( .Z(net1), .A(net2), .B(net3) );
 AND4_2X U2 ( .Z(net1), .A(net2), .B(net3),
.C(net2), .D(net1) );
endmodule
```

As the circuit designer begins to compile the VHDL code, he will control various switches. The switches control parameters such as area, power, and speed. Depending on chip requirements, the circuit designer might decide to compile the VHDL to prioritize only speed, only area, only power, or some specific combination of these interests. Results will vary depending on these priorities, so he inputs these choices to the compiler before it begins.

Drive Strength

The compiler can create nets that are extremely large. There may be hundreds of thousands of cells on one particular net, for instance. The more cells we have on a net, the more power we need to drive them. If we try to drive too many gates from a single source, we might overload our driving transistors. Our circuit will not work.

Therefore, before we can start layout, we need to modify the netlist to make sure that these large nets are adequately driven. To do this, we replace the cells that are driving the net with cells of identical logic function that have larger driving capability. Driving capability is referred to as the **drive strength,** or **fan out,** of the cell. The fan out number indicates how many devices a gate can drive. A driving gate can be any cell in a library.

For example, we might see that our cell library has 10 or 15 different sizes of inverters. These inverter selections might be referred to as *1x, 2x,* or *4x* inverters. These designations on the inverters refer to the drive strength of each inverter. Since a *1x* can typically drive two gates, a *2x* would drive four gates, and a *4x* would drive eight gates.

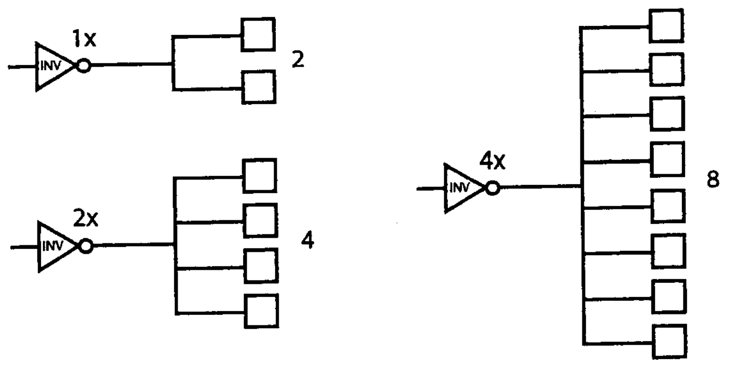


Figure 1–1. One inverter drives two loads, so a 2x drives 4 loads, and a 4x drives 8 loads.

You might wonder why we don't just build one huge gate to cover all circuit eventualities. We could do that. However, we would waste circuit area and burn more power than is necessary. The wisest technique is to be sure that you can drive what you need, and no more. So, during the compilation process, the compiler examines the number of gates on each net and adjusts the size of the gate driving each net accordingly. If the net is too large to be driven by our maximum drive strength device, the compiler will break the net into smaller sections that are easier to drive.

Buffer Cells

If the compiler breaks a large net into smaller, more easily driven sections, it will insert additional gates to drive each smaller newly created net. These extra gates were not part of the original logic. The circuit designer did not add them. You did not add them. The computer made the decision by itself.

These extra gates are called **buffer cells.** Buffer cells help drive gate and wiring capacitance. A buffer cell has no logic function associated with it. Whatever logic signal is fed into the buffer cell appears at its output.

In the next section, we will see an example of how the compiler uses these concepts to drive a large clock net.

Clock Tree Synthesis

Most digital circuits have a clock waveform that clicks away in the background. Every function is synchronized to that click. The wiring nets for this clock timing signal are called **clock nets.** A clock net is usually very large. Typically, the net connects to thousands of gates.

It is impossible to create a cell with enough drive strength to drive all the gates on a clock net, so we have to do some extra work to get the clock net to function. We split the clock net into smaller sections and add buffer cells, as mentioned previously. The net is split into a branching-out pattern, called a **clock tree.** Establishing the tree is called **clock tree synthesis.**

To illustrate how the tree concept works, let's look at a small example. Let's say a certain clock net has six gates on it, and the maximum drive strength our library offers is a fan out of only three. Therefore, we cannot expect one gate to drive the entire clock net. So, we break the net into two smaller sections, and drive each section separately.

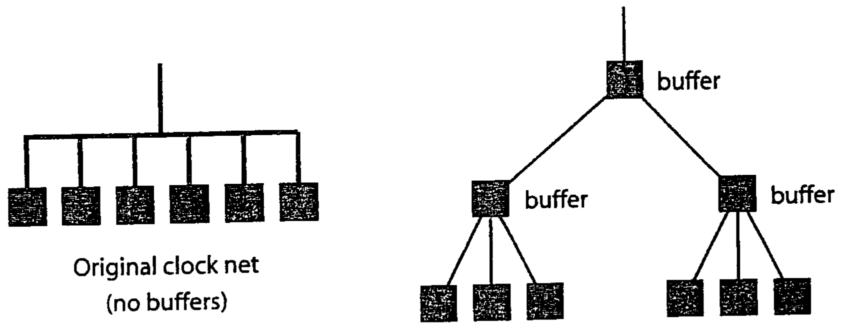


Figure 1–2. Adding buffers to smaller sets of gates to help drive the signal.

You can see in Figure 1–2, that the compiler added two lower level buffers to the circuit, one buffer to drive each set of three gates. The compiler also added another higher level buffer to drive the two lower level buffers. So, three extra cells have been added to our circuit.

If our clock net was even larger, the compiler would continue branching in this manner, splitting the net and adding additional buffer cells, each one driving no more than three others. You can see how this would form a very large tree with many levels.

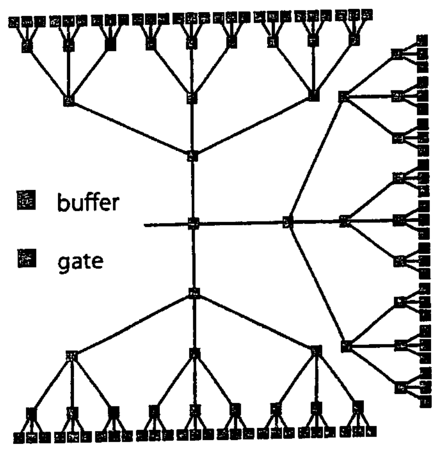


Figure 1–3. Large nets are broken into many smaller sections that can more easily be driven.

With a large number of added buffer cells, the extra cells will introduce extra delays that were not accounted for in the original simulations. Not only that, but other large nets may require this same sort of tree synthesis as well, adding even more buffer cells, also creating delays. Therefore, once the clock net is synthesized, and any other large fan out nets are buffered, we need to re-simulate our design using the compiled net list. Compiling creates a need to re-simulate. This sort of iteration is common in chip development. The good news is that it is not a never-ending story. At some point, you will have a finished netlist.

We are now ready to start the layout process. We begin with floorplanning.

Layout Process

We are now ready to use a suite, or package, of automated software tools called the **place and route tools.** Place and route tools cover the gamut of higher level and lower level software assistance leading to your final layout. As the name implies, these programs generally *place* the gates and *route* the wires, in addition to other helpful functions.

Floorplanning

The first piece of software we will use from the place and route tool suite is called the **floorplanning tool.** It will help you create areas of functionality on your chip, determine the connectivity between these areas, determine your I/O pad placements, and give you feedback on how easy your floorplan might be to wire. The floorplanning tool gets its connectivity and gate information based on the netlist file, created by the compiler software.

Let's follow the floorplanning tool in more detail, beginning with your initial decisions.

Block Placement

Typically, your chip will be divided into various functional areas. For example, if you are working on a large digital chip, there might be a microprocessor unit in your chip (MPU), perhaps a floating point unit (FPU), maybe a RAM block and a ROM block.

Where you locate each area of functionality is your decision, not the computer's. You might say, "Ok, all of the gates for the microprocessor I want in the bottom left hand corner. All the gates for the RAM I want in the top right corner." And so on. You will have a chance to change these decisions later, once you see how your decisions might affect your layout, particularly the wiring.

Case 5:03-cv-04669-JW Document 243-8 Filed 09/10/2004 Page 11 of 27

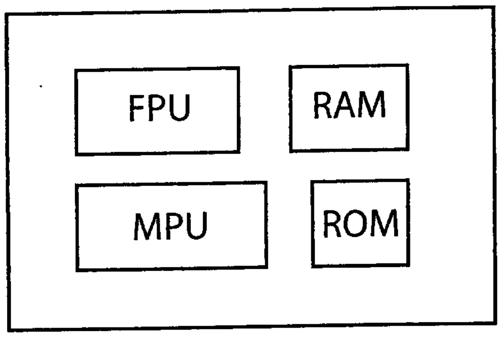


Figure 1–4. Chips have well-defined areas of functionality.

Gate Grouping

Once your areas of functionality are specified, the first task you would want to do is gather together, to some degree, the gates used in each block. You would not want FPU gates scattered throughout the ROM or RAM blocks, for example. Associated gates should all be located near each other.

The floorplanning tool begins by helping you gather your gates together. The exact placement of each gate is not determined at this point. We do not yet need this level of detail. Besides, we might be changing our block placement decisions at some later point. So general vicinity placement is good enough for now.

Block Level Connectivity

Next, your floorplanning tool will help you place the input and output (I/O) cells of your chip. For instance, you would want all the inputs that go to the FPU close to the FPU block in the corner. To help you with this, some tools will actually place the I/O cells in the appropriate areas automatically; other tools will provide graphic feedback for you based on your placement decisions.

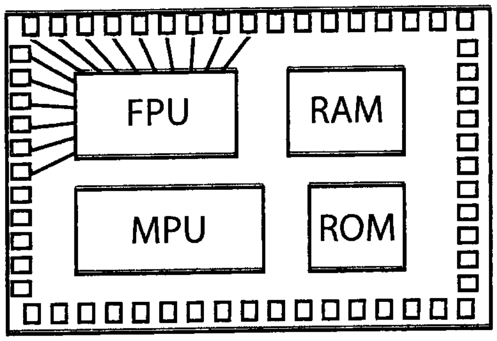


Figure 1–5. Inputs and outputs can be located near their appropriate cell block.

The floorplanning tool also shows basic wiring connections that must travel between blocks. It will show connections between the FPU and the RAM blocks, for example.

The automated software programs such as the simulator and place and route tools make choices a conscientious human with enough time would make, given the same information. (We hope.) However, you can see that constant human intervention and monitoring are essential.

The software never operates without human supervision (you). You have broadly defined where you want your high levels of functionality and your inputs and your outputs. You have predetermined some basic layout instructions for the software, depending on the chip specifications, the size of the final package in which they will be placed, the specific circuitry, and ultimately, on your understanding of how the software operates.

The tools never run completely by themselves. The human brain must oversee the working of the tools or the tools become useless.[2]

Using Flylines

Typically, the floorplanning tool will show you all the wiring lines coming from each block connecting to the I/O pads and to other blocks. All these myriad of wiring lines are what most tools call **rat's nests** or **flylines.**

As you click, drag, and resize blocks around your computer monitor, you will see all these wiring connections moving around in real time with your cursor.

As you drag your blocks around with your cursor, watch the lines. If the lines become badly crossed-over and generally messy, you know that it will be tough to wire the circuitry. If there are no cross-overs of the flylines, then it will be easy to wire.

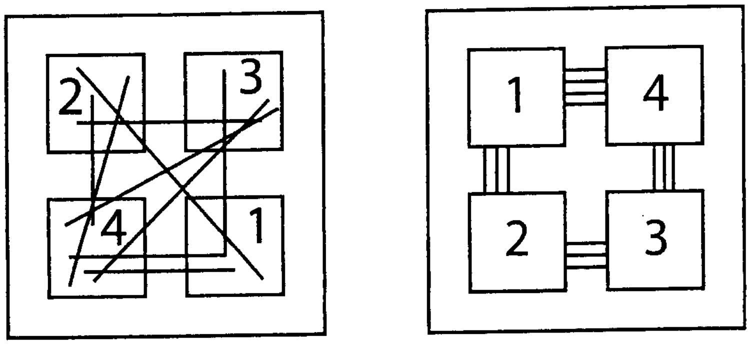


Figure 1–6. Neat flylines indicate good floorplanning.

[2] Just as spell checkers knead a human aye.

You will make changes to your block floorplan so that your rat's nest eventually looks as clean, nice, and wireable as possible.[3] You might decide to relocate entire areas of functionality. You might bring one small block across to the other side and fit it between two larger blocks. You might bring a center block to the outside, or an outside block to the center.

When you finally have a block diagram which gives you nice, simple wiring, you save your floorplanning output files.

Timing Checks

Since the final floorplanning tool output files specify where the gates will be generally located, the placement tool roughly knows how long all the wires will be. These wiring length estimations are based on the physical dimensions of the digital library.

Using this information, your floorplanning tool can output an estimated wire length file that goes back into the digital circuit simulator. You now can run some simulations to determine how your estimated wiring lengths will affect your digital circuit. You must check the possibility that long wires will slow the circuit signals too much, affecting the circuit timing.

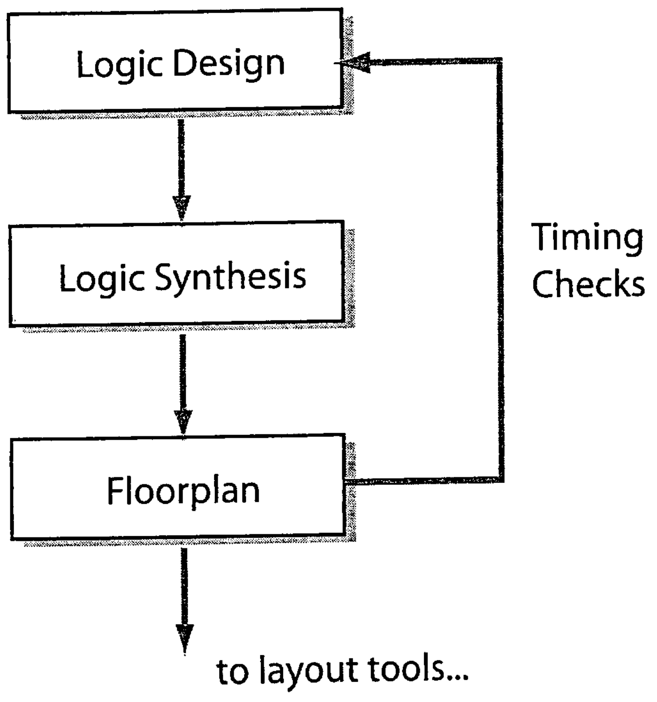


Figure 1–7. The floorplan/timing loop.

[3] As Chris always says, "I don't care if it works, as long as it looks good."

Case 5:03-cv-04669-JW Document 243-8 Filed 09/10/2004 Page 14 of 27

If the wire lengths are indeed overly affecting the circuit timing, the designer will need to modify his design, based on your floorplan. He will change the netlist. He might place higher-powered cells in the block to drive the extra wiring capacitances, for example. As the designer works to better organize the design, not only is it easier to wire, but you will find the chip operates better in the end.

You go around this floorplan and timing check loop a couple of times. The simulator will eventually let you know when you have met the timing criteria.

So, at some point, you decide you finally have a good design. You then move on to cementing your devices in place, so to speak. The fine-tuning now begins that we have been putting off.

Placement

We can now nail down the exact positions of all the logic gates within each block, using a placement tool. In the next section, we will then connect the gates with an accurate and final wiring plan, using a routing tool. The placing tool and the routing tool are two of the many software programs that make up the generally named place and route tool suite.

The task of writing one piece of software to do all the placing and routing functions at once would be enormous. Place and route tools are typically broken into individual pieces of software that address specific areas. One piece of software will perform the placement, one piece will perform the wiring, one the block diagramming, and so on.

Some of the software tools have a nice, fancy front end to them that makes it seem as though one program goes off and renders all the output for the chip at once. But, in the background are still many individual programs feeding their outputs to each other, operating one at a time.

There are various differences in the ways place and route tools appear to the user. However, if you pull back the secret curtain, you will see they all essentially work the same way. It is primarily just the user interface that appears slightly different in each of the various software packages sold.

The placement software starts by selecting one block to work with first. It looks for components associated with the selected block. The tool sees 5,000 gates associated with the floating point unit, for example, and wants to place them.

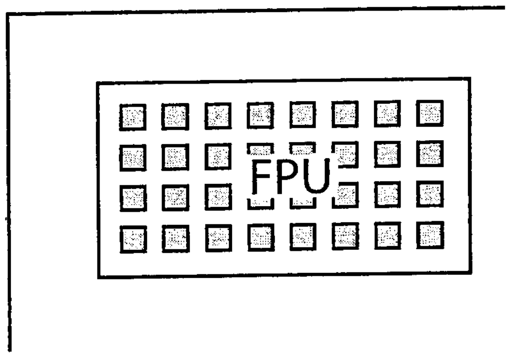


Figure 1–8. Place and route tools place gates in their exact locations.

It might look at the 5,000 gates and basically say, "Well, I see from the floorplan that these 25 gates in the netlist are all to be connected together, so I'll place those gates as close to each other as I can. That way I won't have wires going all over the place." The placement software continues placing logic gates based on their connectivity and the output files from the floorplanning tool.

The initial placement scheme can be considered just a first pass. Depending on the tool, device placement might require multiple passes. The tool says to itself, "Ok, I'll come back and improve the placement, look at it again, improve it a little more, and keep working the placement a little at a time until I get a placement I think is easy to route." The final placement will predetermine much of your eventual layout.

There are placement tools available that can make gate placement decisions based upon the signal timing of a design. This placement approach is known as **timing-driven layout** and has quickly become standard practice. The end result is usually far superior to more traditional techniques.

I/O Drivers

Not only are your gates inside the chip placed at this time, but all the **I/O drivers** are placed at this time, as well. These I/O drivers are the special cells that will drive the input signals, provide outputs, contain ESD protection and test circuitry.

These drivers are placed using separate placement tools that know about the I/O rules. They place the I/O pads separately from the placement of the standard logic cells.

Finally, after all these automated tools and all the timing feedback looping, you have the best placement you think you can reasonably make. Next you will route the wires.

Routing

With your gates and I/O cells nailed in place, you will now start to wire everything together. You will take the file that popped out of the compiler, and drop it into another software program. Again, we rely on automation.[4]

Your wiring software has two priority nets—power and clock signals. It will route these two types of nets first since they are the most critical.

After the power rails and the clock signals have been placed, your wiring software will continue to wire the remainder of the circuitry, beginning with any other circuitry you declare as critical.

We will next examine these specific wiring concerns, in order of importance, beginning with the power nets.

Power Nets

There are certain rules for connecting power to logic gates. Wiring must be centered in certain places and run in certain directions. You end up with power rails running through the middle of your gates, as in Figure 1–9.

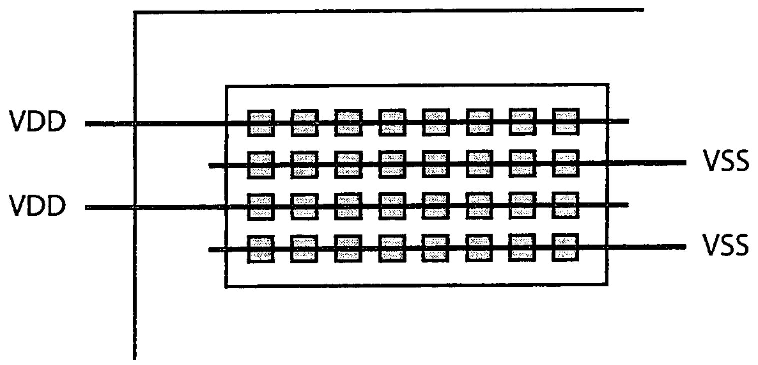


Figure 1–9. *Power rails running through our gates.*

You can intervene in the automated wiring at any time, of course. Perhaps you want to do something special with your power wiring. Maybe you want to move certain blocks around. Maybe you want to put extra power wiring in a particular portion of the circuit because you know the area will demand more power under certain circumstances. It helps if you understand the function of your circuit so that you can make these decisions.

[4] We would not be able to automate so much of our digital layout process without the standardization techniques found in Chapter 2. While these are options available for analog layout, standardization techniques are absolutely required in the digital world. See Chapter 2. (In fact, see all the chapters. After all, you paid for the whole book.)

The wiring software is driven by the net list, which is aware of every component. Therefore, it can tell you when it has completed wiring the power nets.

Strapping

In Figure 1–10, notice the highlighted cell at the far upper right corner, farthest from the VDD input pad. As the power comes into the circuit on the lower left, it must travel through the existing rails. You can see that the supply current has a lot more metal in series to go through to get to our little end cell in the upper right corner.

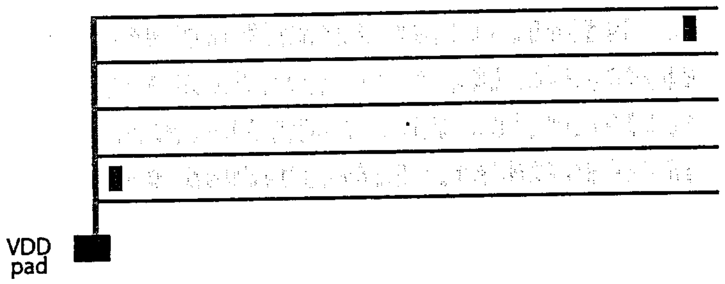


Figure 1–10. *Distant cells see more resistance through the rails due to the length of wire. If only we could lower the resistance to those remote cells.*

The other highlighted cell nearest the pad would see a lot less resistance, being so close. Let's see how we can alleviate this difference.

By laying straps of metal across your power rails, you create a big waffle iron grid of multiple paths. Now it's like having resistors in parallel. The overall resistance reduces.

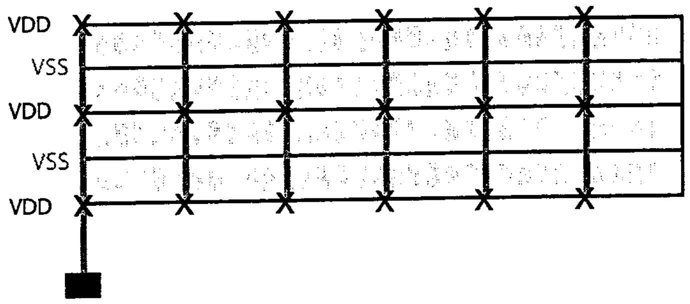


Figure 1–11. *Strapping our rails.*

We have given the current multiple paths. We just blanket the chip with straps and rails to give the chip the most parallel routes for power as possible. The more the merrier.

The older power routers had a uniform spacing rule for power strapping. The power straps were placed every so many microns, regardless of need. You would just say, "Oh, I'll have a metal strap every 60 microns or every 100 microns." Perhaps you thought these were good average numbers, so you would just go with it.

However, newer tools actually look at the drive strengths in the cells before they place their power straps. The router can calculate, for example, that there is a large concentration of huge drive strength cells in certain areas that require more power. So, the router places more power strapping in that area.

Power net wiring is more important than it used to be. More intelligence is being built into the tools to know more about what's going on in the circuits, to give us more intelligent auto-placed and auto-routed chips. It's taking some of the mystery out of the process.

Digital mask designers need to know more about their tools and techniques than they did before we had so much sophistication. They have to understand why the tools are doing what they do. There is more need to understand when the tool messes up, misinterprets, fools itself, or gets stuck. At times we must override the tool's choices. Rules are getting intricate. Tools are getting intricate.

For example, a mask designer might say, "Ok, the power router hasn't done a particularly good job around here. I know this is a very power hungry block, so I'm going to have to go in here and manually put in some extra power strapping."

There can be a lot of manual intervention with digital layout. Using the tools, rebelling against the tools, improving what the tools have routed, finding work the tools left behind, and giving the tools some human touches. People drive the tools.

Clock Net Wiring

Once you have finished running the power rails, your software usually offers a specialized tool just to wire all the clock nets, the circuitry distributing the timing signal. Remember the clock tree synthesis we generated earlier? This is a very critical net.

There are various approaches to wiring a clock net. Each tool has its own particular way of operating.

If you are unfortunate enough to have to hand wire a clock net, you can use the **Central Clock Trunk Approach.** There is usually a clock driver cell that has enough drive strength to drive the top level clock buffers. Place that cell centrally within your design and create a large central trunk that branches out to join to all the clock buffers.

As the net reaches further out from the main driver, it continually splits into more and finer branches. The wire widths at the outer edges become smaller and smaller. The large central region resembles a thick tree trunk, hence the name *Central Clock Trunk Approach.*

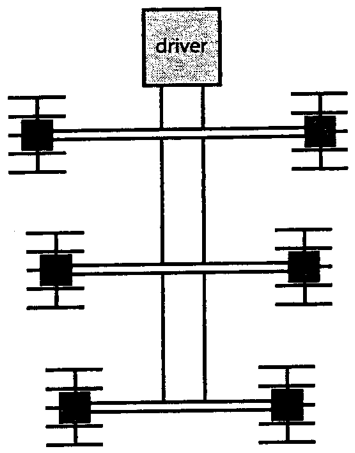


Figure 1–12. Central Clock Trunk Approach to wiring clock nets.

This Central Clock Trunk Approach is very easy to wire. The clock signal distributes very easily.

Other Critical Nets

At this point, we turn to any other nets that need special attention. You wire the nets that you are most worried about first, maybe by hand. You should have been given a list specifying the critical nets of the chip. (If not, ask.)

You plug the critical nets file into the auto-router and it goes off, routing the critical nets while wiring is still relatively easy, before we wire the bulk of the chip. As always, you can intervene at any time until all the critical nets are exactly what you want.

Remaining Nets

The last thing you do is wire the rest of the circuitry. If you are lucky, the tool will know how to automatically wire everything else on its own. Then it says, "I'm done. I'm finished."

Finishing the wiring can take days on really big chips. Typically, the software will go away and work silently on its own. You just sit there and play backgammon. Then just as you discover that elusive guitar chord you've been trying to find, the tool beeps and says, "Finished."[5] Seriously though, there is usually plenty of time to do more productive things while your software is running.

Finishing the Wiring by Hand

When the auto-router finishes as well as it can, you might end up with all the wiring hooked up on your chip. Or, more likely, you might not. Usually, the routers just cannot get to some areas. It can wire itself into some blind alleys and helplessly sit there thinking it is finished. You look at the auto-router's work and ask it, "Ok, how many of the nets couldn't you do?"

Usually you will have to use the human eye, break some nets, and move stuff around, in order to complete the wiring. You can count on some final intervention on your part to finish all the nets and do all the wiring the computer could not do.

For example, the router might say there are five nets it just could not complete. It will highlight for you where these open nets are located. Zoom into the area where the pin is located that it says it cannot wire. Usually in these cases, you will see that the router has placed the wire close by, but it stopped because there is a bunch of stuff in the way. Usually it's something really obvious—a bunch of nets that were pre-wired earlier may be in the way, for example.

Nine times out of ten it's pretty straightforward to move some wires out of the way or do a little re-wiring to free up some area. Then you can normally run the wire where you need to.

It's unlikely the auto-router will do 100% of the wiring in one pass.

With experience, you will often manually pre-wire some signals because you know the auto-router will have problems with them. It saves a lot of time as you learn to preempt these problems in advance.

There are some times, if the chip dimensions are too small, you may not be able to wire the chip at all. You just cannot place 5,000 wires in a tiny 100-micron space.[6] Or, you might have lots of room, but there might be just too many crossed nets, or a bad floorplan. For whatever reason, sometimes you

[5] Chris has a thank you on our band's CD to a certain tool that allowed him the time to practice his guitar licks at work. It's a tool with a reputation for being slow. I'm not sure if they still use that tool, but Chris' guitar licks are still hot and polished.

[6] Yet.

just cannot do it. In that case, start over. Go back to your floorplanning and begin from scratch.

Eventually, you really are finished. All the gates are wired. At this point, you have the computer spit out a wiring file that says, "Ok, here are the real wire lengths, and the real wire capacitances." You have them in hand. No more fine-tuning. No more estimating. This is the real stuff.

Prefabricated Gate Array Chips

All of the above techniques will also be used on a special type of chip, called a **gate array.** A gate array is a predefined and partially prefabricated chip, literally an array of gates. The semiconductor manufacturer processes wafers up to just before the metal is deposited, then stops and stores them until needed.

You will still use floorplanning tools, placement tools, and wiring tools. However, you are not placing any diffusion or poly, only metalization and contact layers. This type of chip is useful for prototyping circuits. Instead of waiting twelve weeks or more for your chip to be fabricated, the processing time for a gate array only takes a few weeks, since you are only fabricating the back end of the process.[7]

Typically, you can choose from only a few fixed-sized gate array die—small, medium, and large. The manufacturer tells you a certain gate array will handle, say, 5,000 gates, this other one will handle 10,000 gates, and this last one will handle 50,000 gates. You select from the offered sizes.

If you have a 30,000-gate design, but your supplier does not have a gate array designed for 30,000 gates, you have to move up to the next size, the 50,000-gate array. You waste a lot of space. Also, the 50,000-gate might have provision for 150 I/O pads, but your design only needs 50 I/O pads. So, with all this extra baggage on the gate array, you end up with a much bigger chip, but you get them built quicker.

When you have finished your logic design, are happy with it, and you have proven your concepts using a gate array, you then convert the design into a real fixed device, a full-custom chip that goes into production.

[7] This reminds me of the old cardpunch days, taking a week to see how my Fortran programs turned out. And I'm talking a week for each attempted fix to the program. Imagine. You kids don't appreciate how we trudged ten miles in the snow, marched thousands of cold cement steps to the dark basement, to drop off a batch job at the one room-sized computer on the UC Davis campus. (Well, it snowed in the California central valley pretty bad in those days.) Now you can make a whole chip in that time.—*Judy*

Verification

It's time to verify that what the team originally had in mind is what was actually built. You will now verify the design.

Design Verification

You feed your new wiring file back to the simulation people again. This time they simulate with the actual wire data. No more estimation. The wiring is physically in place this time.

If you are lucky, everything's happy. If your tools are good enough, and your models are good enough, then the wiring will be fine. If you are not lucky—for instance, the router has done a bad job, or you have done a bad job—the circuit designers may have to change the design again.

If you need to redesign, it is not necessarily a total loss. The team may be able to keep some of the original work—just merge in some new logic, rip out some cells, replace some cells, and re-wire. However, there are times you might have to actually redo the whole enchilada.

The digital mask designer makes these changes. It's a matter of running the tools and software. You typically do not pull out one or two little gates and replace them by hand. The software is intelligent enough to merge in new connectivity and do the majority of the restructuring.

As you can see, a good layout person has to know a lot about the tools and what they are trying to do in the circuit in order to be successful. A good mask designer can put a good floorplan together from the start, especially if they know some details about how the circuit functions. They can preempt problematic issues that they know might develop.

Digital layout is a skill. It is not just a case of pressing the buttons and playing backgammon all day.

You have to know where the tools are going. You have to foresee issues that cause the tools to stumble, and work in advance to prevent those issues. You have to nurse the chip through these processes. You have to direct the flow.

As you master good layout skills, make them part of your daily habit. Soon it all just seems to happen by itself.

Eventually, when all the timing is done, all the wiring is placed, and the chip has been re-simulated, everyone is happy. You have finished the top-level layout of your chip. You have converted a database from a conceptual format into a real mask design.

Physical Verification

Up to this point, you have not been working with real transistors. You have just been working with little boxes with points on them that say, "This is the input. Here is the output. We do not care what is inside." There are no real transistors in those boxes.

GDSII File

To complete your mask design, you take this abstract, high-level database, and replace the boxes with the real logic gates. You merge the real components from real libraries with the wiring and placement data from the place and route tools. As you replace the abstract components with real library components, you produce a **GDSII** stream file of your chip. This is a file that has all the components, all the wiring of your cells, all your vias, everything.

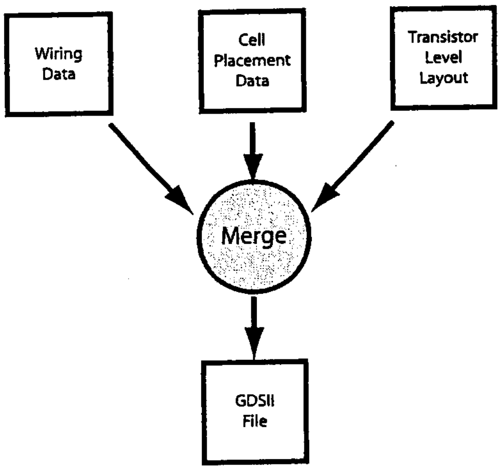


Figure 1–13. The data is merged with the actual transistors to produce a GDSII stream file.

DRC and LVS Checks

By the time you produce your final GDSII output file, the chip has gone through the wringer. Typically, there have been so many operations performed on these databases, from start to finish, that you no longer trust the final output. Has the software kept up with all the loops and changes well enough? Those nice software people do make mistakes, you know, especially when we run lengthy and complex iterations on a project.

Case 5:03-cv-04669-JW Document 243-8 Filed 09/10/2004 Page 24 of 27

Once you get your GDSII stream file, you then will want to run checks to be sure that the wiring is correct and complete. At this point, we check all the process design rules using Design Rule Check (DRC) software.

At the same time, we also check that the wiring and transistor connectivity correctly match the connectivity defined in our netlist. We use Layout Versus Schematic (LVS) software to perform this connectivity check.

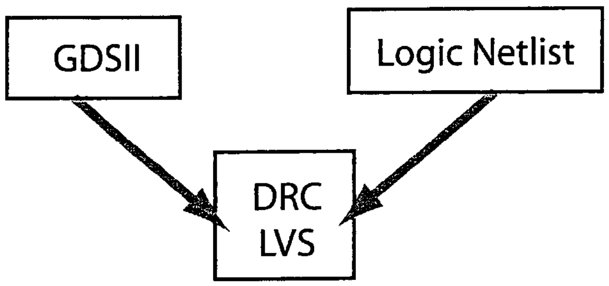


Figure 1–14. *How did we do? Let's run a check.*

Library Management

What could possibly go wrong in this highly automated, computer-controlled nirvana? Plenty.

The place and route process relies on synchronization of all the various database files for all the tools. It relies on the fact that the layout for your inverter is correctly represented by the schematic for the inverter.

Having various representations of the logic gates creates very complicated library management issues. For example, you could have a cell called *Inv1x*, but it might not LVS to the schematic of *Inv1x* because someone has made a mistake in the compiling of the library.

As another example of a possible problem, the place and route tool might not have up-to-date information. The place and route tool might think everything is fine according to what it knows. However, when you merge your cell layout data with the wiring data you could find wires just dangling in free space. The problem could be, for example, that the representation of the inverter the place and route tool uses shows a pin one grid lower than the representation in the GDSII file.

Good library management for digital place and route is very important. An incremental change to one cell in a library could cause six or seven files to require updates in multiple directories. There are so many files, which have

DEF085326

to be 100% perfect and synchronized with each other, mistakes are easily made.

However, if your tools are good, flows are good, and libraries are good, your design should pop out DRC and LVS clean, first time out of the box. If not, you have a bit more work to do. We have a section later in the book to help you resolve these errors. It is not always as difficult as it may first appear.

Summary and Flowchart

So, there you are. That's how large digital chips are created. You have created your netlist. You have placed your gates and routed the wiring. You have considered critical elements of your circuit. Finally, at some point, you are finished.

Tape out. Clean chip. Pass Go. Collect $200.

That was really something, wasn't it?

I feel a flowchart coming on... Yes, let's look at all that again as a flowchart.

The flowchart represents the steps of a typical digital layout process. Follow along in the chart as you read each paragraph below. (See Figure 1-15.)

First you design your logic, synthesize it, draft a floorplan, and then do some timing checks around that loop for a while.

Then you run your place and route tools and do some timing checks. You keep going around that loop until you are happy. You may even have to go around the floorplan loop a couple of times again.

At this point, you need the digital libraries. By library I mean the AND's, OR's, input cells, output cells, and all the real transistor level components. You need the digital libraries to make your final GDSII file.

Finally, you run the DRC and LVS checks against the netlist you started with, out of your logic synthesis. Then you get the final chip done.

You can see from the massive use of computerized tools, that your digital library devices must be put together with a lot of standardization. Your cells, your wiring, and your logic must all be pre-designed with computerized placement in mind.

Analog mask designers get to use standardization techniques as options when appropriate for their projects. However, digital mask design absolutely

Flowchart of Digital Layout Process

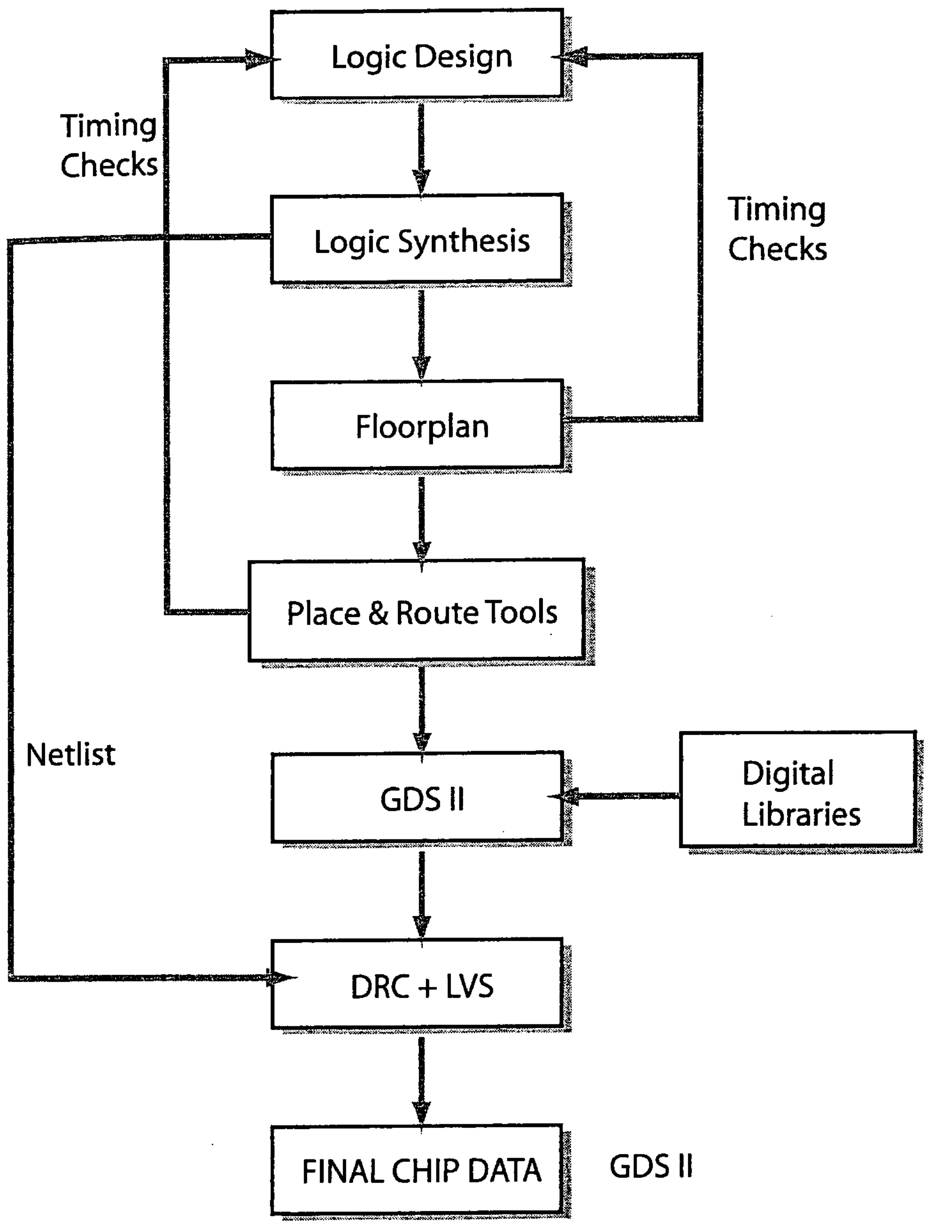


Figure 1–15. Our original design undergoes many iterative loops and manual interventions before we have our final data for the chip layout.

demands use of standardization techniques. Automated software tools would not work without it. That's where we go in the next chapter.

Closure on Digital Layout

As you can see, automated digital layout can be very complicated and convoluted. In reality, though, all we are doing is using very simple concepts, just on a massive scale.

Most companies that use these tools to design large digital chips also have very well documented procedures that are designed to take you through every stage of the layout process. There are so many feedback paths and decision points in a large digital design that it is otherwise impossible to keep track of where you are in the flow. You can use the procedures to help you take the project a step at a time.

On the face of it, digital layout may appear as though we are just turning a handle and spitting out fully finished chips. However, there is a lot of manual intervention required. The mask designer can stop the process, make adjustments, preset priorities, work ahead of the automation to avoid problems before they happen. The more you know about the automated procedures, the more effective you will be at the helm.

Here's What We've Learned

Here's what you saw in this chapter:

- Place and route tools
- Floorplans and flylines
- Priority nets
- Feedback loops in the design and layout process
- Troubleshooting automated procedures
- Placing buffer cells according to drive strengths
- Gate arrays
- GDSII from place and route tool
- Netlist from logic synthesizer tool
- DRC and LVS checks
- Flowchart of digital layout procedures